

HDL-TM-88-13

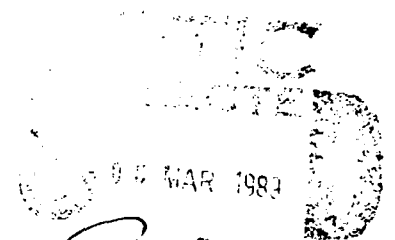
February 1989

4

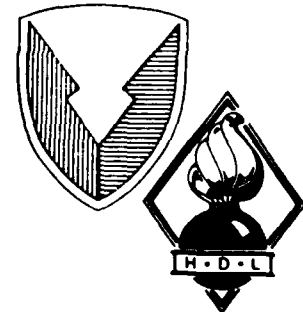
ADP-1905 268

Implementing a Definite Clause Grammar for Parsing Surface Syntax

by John O. Gurney, Jr.
Kimberly C. Claffy
Jason H. Elbaum



A



U.S. Army Laboratory Command
Harry Diamond Laboratories
Adelphi, MD 20783-1197

Approved for public release, distribution unlimited

82 3 03 139

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturers' or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION / AVAILABILITY OF REPORT		
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE			Approved for public release; distribution unlimited.		
4. PERFORMING ORGANIZATION REPORT NUMBER(S) HDL-TM-88-13			5. MONITORING ORGANIZATION REPORT NUMBER(S)		
6a. NAME OF PERFORMING ORGANIZATION Harry Diamond Laboratories		6b. OFFICE SYMBOL (If applicable) SLCHD-TA-AS	7a. NAME OF MONITORING ORGANIZATION		
6c. ADDRESS (City, State, and ZIP Code) 2800 Powder Mill Road Adelphi, MD 20783-1197			7b. ADDRESS (City, State, and ZIP Code)		
8a. NAME OF FUNDING / SPONSORING ORGANIZATION U.S. Army Laboratory Command		8b. OFFICE SYMBOL (If applicable) AMSLC	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER		
8c. ADDRESS (City, State, and ZIP Code) 2800 Powder Mill Road Adelphi, MD 20783-1147			10. SOURCE OF FUNDING NUMBERS		
			PROGRAM ELEMENT NO. P611102.H44	PROJECT NO. AH44	TASK NO.
					WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) Implementing a Definite Clause Grammar for Parsing Surface Syntax					
12. PERSONAL AUTHOR(S) John O. Gurney, Jr., Kimberly C. Claffy, and Jason H. Elbaum					
13a. TYPE OF REPORT Final		13b. TIME COVERED FROM Oct 86 to Sept 87		14. DATE OF REPORT (Year, Month, Day) February 1989	
				15. PAGE COUNT 83	
16. SUPPLEMENTARY NOTATION HDL project: AE1754, AMS code: 611102.H440011					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD GROUP SUB-GROUP			Definite clause grammar, parsing, syntax, natural language, computational linguistics		
05 07					
09 02					
19. ABSTRACT (Continue on reverse if necessary and identify by block number) ✓ With this report we aim to serve several ends: We acquaint interested readers with some simple techniques for processing natural language (English) sentences. We present basic ideas and document our implementation of a software environment for building parsers that use these techniques. We illustrate the use of the implementation by recounting the development of an elementary (or naive) parser in the form of a definite clause grammar. We discuss choices we have made and some of the issues considered in making these choices. We also mention some of the possible extensions to the grammar.					
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a. NAME OF RESPONSIBLE INDIVIDUAL John O. Gurney, Jr.			22b. TELEPHONE (Include Area Code) (202) 394-4300		22c. OFFICE SYMBOL SLCHD-TA-AS

DD Form 1473, JUN 86

Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

Contents

1	Introduction	1
2	What is a Definite Clause Grammar?	2
3	Implementation	5
3.1	Prolog	5
3.2	Left Recursion	7
3.3	The Structure Argument	9
3.4	Terminal Rules	10
3.5	Interfaces	10
4	Building a Grammar	12
4.1	Governing Considerations	12
4.2	General Constraints	13
4.2.1	Number Agreement	14
4.2.2	Pronoun Case	15
4.2.3	Transitivity	16
4.3	Phrase Structures	17
4.3.1	Noun Phrases	17
4.3.2	Higher Order Structure	19
4.3.3	Subordinate Adverb Clause	19
4.3.4	Sentence Prefixes	21
4.3.5	Verb Tenses	22

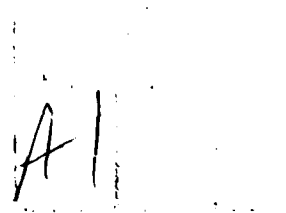
5	Ground Not Covered	27
6	Concluding Comments	30
	Literature Cited	33
	Distribution	83
	Appendices	
A	Simple Grammars	35
B	PROSENT	39
C	PROTREE	43
D	The Grammar	55

List of Figures

1	Two parse trees.	3
2	Prolog window showing sentence queries.	6
3	Infinite loop from using left recursive grammar.	8
4	Bindings for structure arguments returned by Prolog.	9
5	Mouse sensitive tree.	11
6	Noun phrases and higher structure.	18
7	Infinitive verb phrase as subject.	20
8	Subordinate adverb clause.	21
9	Leading adverb phrase.	23
10	Tensed verb.	26

List of Tables

1	Verb Tenses	23
---	-----------------------	----



1 Introduction

With this report we aim to serve several ends: We acquaint interested readers with some simple techniques for processing natural language (English) sentences. We present basic ideas and document our implementation of a software environment for building parsers that use these techniques. We illustrate the use of the implementation by recounting the development of an elementary (or naive) parser in the form of a definite clause grammar. We discuss choices we have made and some of the issues considered in making these choices. We also mention some of the possible extensions to the grammar.

The grammar we present amounts to a modularized syntactic parser which can produce parse trees for some of the well-known forms of English sentence. In choosing rules for the grammar, we have concentrated on certain noun phrases, verb tenses, and the subordinate adverb clause, and a few other constructions.* All sentences that we consider are declaratives. These selections reflect our interest in certain problems involving recovering and regimenting information from extended discourses.

A possible application of our ongoing studies would be autonomous processing of streams of natural language (or other richly formatted) messages (as well as briefing materials and texts) about events and situations. This processing would employ modules based on pragmatic and semantic knowledge. The grammar we present here does not employ pragmatic or semantic knowledge and, therefore, amounts to a syntactic module. It delivers syntactic parse trees intended as input to, or material for, the other modules. The kinds of message and text we have in mind convey information about where objects are located in time and space, as well as information about the course of events in time. And this information is largely and essentially relational (e.g., it says that A happened **before** B; that C is **in front of** D). Natural language handles these kinds of relations through various syntactic devices such as verb tenses, subordinate clauses, and prepositional phrases.

Under current study are two questions: What kinds of parse trees should the grammar deliver? What forms should the next (or other) stages (or modules) of processing take? For these studies, we have found that definite clause grammar is a useful research tool.

*Almost all of the syntactic constructions that this grammar handles were taken from a student grammar handbook (see Warriner [8]).

2 What is a Definite Clause Grammar?

A definite clause is a horn clause in any of the standard logical calculi. An example in the notation of the first-order predicate calculus is

$$(\forall x)(man(x) \wedge unmarried(x)) \Rightarrow bachelor(x).$$

An example from the propositional calculus is

$$P \vee Q \wedge R \Rightarrow T.$$

And an example from the computer language Prolog is

$$bachelor(X) :- man(X), unmarried(X).$$

We can generalize from these examples, stating that a definite clause in logical notation is any clause with only one term on the right-hand side of the conditional arrow \Rightarrow . In the notation of Prolog there must be only one term left of the turnstile $:-$. This definition allows degenerate cases where there are no terms on the antecedent side of the arrow or turnstile. These special definite clauses are unit clauses like

$$unmarried(John).$$

In what follows we will explain how rules of English grammar can be encoded into Prolog using nothing but definite clauses.

Instructive discussions of definite clause grammars (DCG's) appear in Pereira and Warren [5], Clocksin and Mellish [1], and Winograd [9]. Throughout the remainder of this section we review some of the well-known features of these grammars.

Although DCG's have been used for various jobs in computational linguistics, perhaps the most obvious use is to validate that a natural language sentence is grammatically acceptable and then display the syntactic structure of that sentence. As examples of what a DCG can do, figure 1 displays, in graphical form, syntactic parse trees for two English sentences.

A set of grammar rules strong enough to generate these sentences could be the following (see Winograd [9], p. 93):

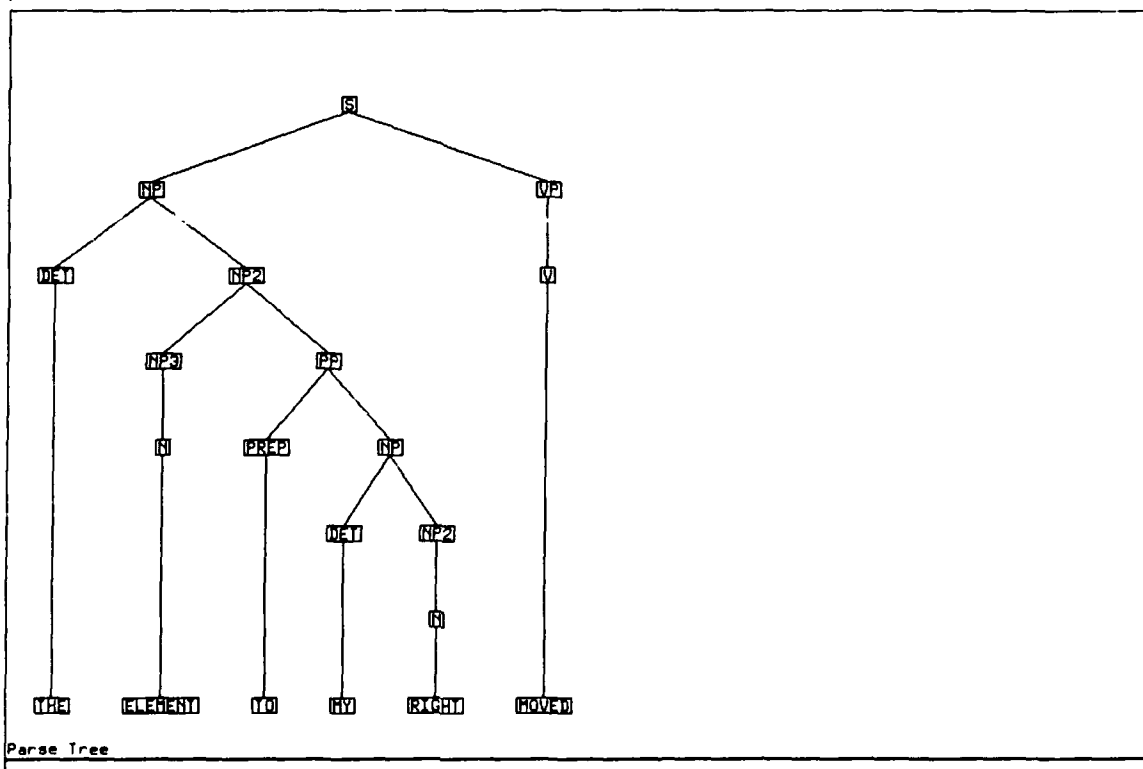
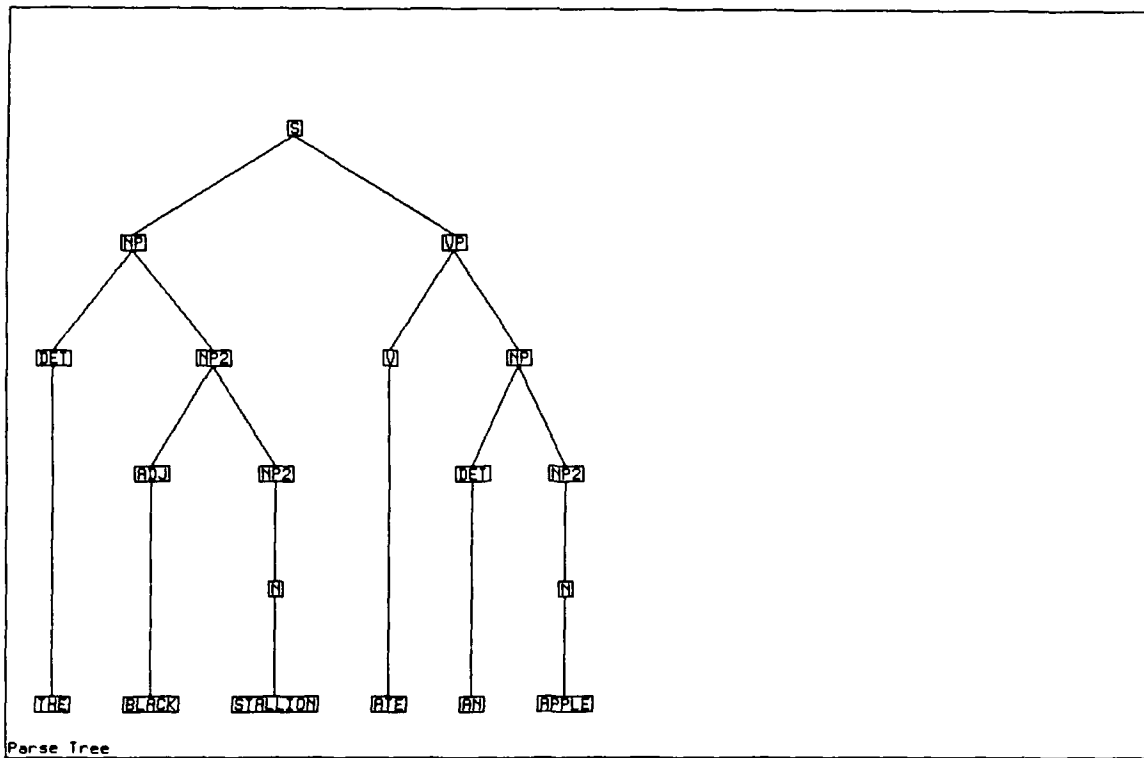


Figure 1: Two parse trees.

$S \rightarrow NP, VP$
 $NP \rightarrow \text{Determiner}, NP$
 $NP \rightarrow \text{Noun}$
 $NP \rightarrow \text{Adjective}, NP$
 $NP \rightarrow NP, PP$
 $VP \rightarrow \text{Verb}$
 $VP \rightarrow \text{Verb}, NP$
 $VP \rightarrow VP, PP$
 $PP \rightarrow \text{Preposition}, NP$

where

S means sentence,
 NP means noun phrase,
 PP means prepositional phrase, and
 VP means verb phrase.

The rule $S \rightarrow NP, VP$ can be read as *One form of sentence consists of a noun phrase followed by a verb phrase*. Alternatively, and more pertinent to definite clause grammar, the rule can be read as *If a sequence consists of a noun phrase followed by a verb phrase it is a sentence.** Thus, the rule can be rewritten as a formula in predicate logic:

$$(\forall x)(\forall y)(\forall z)(noun_phrase(x) \wedge verb_phrase(y) \wedge concat(x, y, z)) \Rightarrow (sentence(z)).$$

where $concat(x, y, z)$ means that the sequence of words, z , is the same as sequence x followed by sequence y . From this form the rule can be readily translated into the formal language of a logical theorem prover such as Prolog. We would then have an implementation of the DCG in Prolog (or whatever other theorem-proving language). Given such an implementation, we could apply the set of grammar rules to any candidate for sentence by asking the logic system to prove that the candidate is a sentence. In the present case, we might ask the logic system to prove

$$(\exists z)(noun_phrase([the, black, stallion]) \wedge verb_phrase([ate, an, apple]) \wedge concat([the, black, stallion], [ate, an, apple], z)).$$

*We make no claim that this grammar is correct for English. In fact, counterexamples to the first rule come easily to mind.

Two features of DCG's make them attractive for our project. First, the coded versions of the rules of the grammar are perspicuous. In Prolog, the rule for sentences (above) looks like the following horn clause:

```
sentence (Z) :- concat(X,Y,Z), noun_phrase(X),  
verb_phrase(Y).
```

The Prolog query looks like

```
?- sentence ([the,black,stallion,ate,an,apple]).
```

Second, the logic system takes care of some of the procedural, algorithmic, and control problems. These systems validate the sentence query by backchaining and unification (see Clocksin and Melish [1], and Pereira and Warren [5]). With these methods, we can quickly develop and modify grammatical parsers simply by adding, retracting, and changing grammar rules in a file (which will be compiled as a Prolog database).

3 Implementation

3.1 Prolog

All of our Prolog implementations used a version of DECsystem-10 Prolog running on a Symbolics 3675 Lisp machine (see User's Guide to Symbolics Prolog [7]). This is the Prolog software offered by Symbolics, Inc. This environment supports calling Lisp programs from Prolog as well as calling Prolog programs from Lisp.

This version of Prolog also supports writing rules in a standard grammar-rule notation. The Prolog rule for sentences (above) would appear in this notation as

```
sentence -> noun_phrase, verb_phrase.
```

This notation is superior to the standard horn-clause notation for two reasons. The rules are now even more readable. For simple grammars, the rules

look very much like the standard Backus normal form used by programming language designers as well as some linguists (see Winograd [9], chapter 3). Also, hidden from view is the fact that the underlying form of these rules dispenses with the **concat** predicate in favor of difference lists (see Sterling and Shapiro [6]).

Using **concat** to ensure that the noun phrase and the verb phrase are contiguous (with the latter following the former) is inefficient. **Concat** can cause much backtracking, which is avoidable with difference lists. However, incorporating difference lists into horn-clause style grammar rules is tedious, resulting in rules that look cluttered. Since the requirements for difference-list variables are simply stated, these variables are generated by an algorithm that translates the grammar notation into the horn-clause notation. The horn-clause notation for the above rule turns out to look like

sentence(S1,S2) :- noun_phrase(S1,S3), verb_phrase(S3,S2).

The Prolog (grammar notation) version of the grammar on page 5 appears in section A.1 (see app. A). This grammar will validate, as grammatical, the two sentences given above in figure 1 (see fig. 2). It will also fail to prove the sentence

the stallion black ate an apple.

We may take this as meaning that the sentence is ungrammatical. In general,

```
?- sentence ([the , black , stallion , ate , an , apple] , []).  
yes  
?- sentence ([the , element , to , my , right , moved] , []).  
yes  
?-
```

Prolog Listener 4

Figure 2: Prolog window showing sentence queries.

we cannot take *not proven* to mean *ungrammatical*, for the obvious reason that our DCG may not have complete coverage of the actual grammar of English. Our simple grammar cannot prove

the black stallion quickly ate an apple

because it contains no rule for adverbs. On the other hand, it can prove

the black stallion beckoned apple

which is not grammatical. Our DCG would require extensive modifications and additions before we could use the provability of a sentence to be a necessary and sufficient condition for its grammaticality. A final grammar including all modifications and restrictions can be thought of as a distant goal toward which we work. Short of this goal, we would produce special-purpose grammars which may be interesting and useful in certain applications. Some of the necessary enhancements to the DCG are discussed in section 4, on building a grammar.

Two problems emerge because our grammar is implemented in Prolog: the problem of left recursion and the problem of building parse tree structures.

3.2 Left Recursion

The grammar in section A.1 is left recursive. For example, in the rule

noun_phrase -> **noun_phrase**, **prepositional_phrase**.

the head predicate occurs as the first clause on the right side. During backchaining, the system could enter an infinite loop. In fact, the query



?- sentence ([the,black,ate,an,apple],[]).

will cause such a loop (see fig. 3). Since *black* is not a noun in the dictionary for this grammar, the above **noun_phrase** rule will call itself recursively. In

```

?- sentence ([ the , black , ate , an , apple] , []).
Error: The control stack overflowed.

Rebinding the following specials; use Show Standard Value Warnings f
or details:
  *READTABLE*

(:PROLOG-PREDICATE NOUN_PHRASE 2):
s-R, :      Continue with a larger stack.
s-B, :      Return to Prolog Top Level in Prolog Listener 4
→

Prolog Listener 4

```

Figure 3: Infinite loop from using left recursive grammar.

order to prevent these loops, one technique has been to alter the grammar, for example, by naming different types of noun phrases (`noun_phrase_2`, `noun_phrase_3`, etc.). We will then have new rules such as

`noun_phrase -> noun_phrase_2.,`

`noun_phrase_2 -> noun.,`

and so on. The revised grammar is in section A.2.

For this grammar, the query

`?- sentence ([the,black,ate,an,apple],[]).`

returns *no*. This grammar seems to look less clear than the former version. We have, in effect, used a trick to avoid a procedural control problem. Or to put it more positively, we have used a declarative representation of possible grammatical structures, and these structures can only be trees (graphs with loops will not exist). Since we may think that the syntactic structure of a sentence can only be some tree or other, perhaps these new nonterminal nodes, `noun_phrase_2` and `noun_phrase_3`, really should belong in the correct or final grammar for English.

3.3 The Structure Argument

The grammars in sections A.1 and A.2 are properly called recognizers because they only perform a test. They do not return any information on the structure of a sentence. Again, there is a fairly standard way to change a DCG recognizer into a DCG parser. We add argument places to the various phrase-structure predicates and rewrite the grammar rules incorporating structural variables which must be bound in order for a proof to succeed. In this way, the sentence rule becomes

```
sentence(s(NPhr,VPhr)) -> noun_phrase (NPhr),  
verb_phrase (VPhr).
```

The new grammar appears in section A.3.

Now successful sentence queries will return a representation of the parse tree in list form rather than simply *yes*. This is because Prolog always returns bindings for all variables that appear in the Prolog query. In figure 4, the queries must now include a new variable (for example, S) in the argument list for the predicate `sentence`. The binding returned for S is the desired syntactic parse tree (in list form). The parse trees displayed in figure 1 were generated by this grammar.

These structures are isomorphic with the clauses on the right-hand side of the rule. For example, in the first rule, the structure argument, `s(NPhr,`

```
?- sentence (S , [the , black , stallion , ate , an , apple] , []).  
S = (s (np (det the) (np2 (adj black) (np2 (n stallion)))) (vp (v at  
e) (np (det an) (np2 (n apple)))))  
yes  
?- sentence (S , [the , element , to , my , right , moved] , []).  
S = (s (np (det the) (np2 (np3 (n element)) (pp (prep to) (np (det m  
y) (np2 (n right)))))) (vp (v moved)))  
yes  
?-  
?-
```

Prolog Listener 4

Figure 4: Bindings for structure arguments returned by Prolog.

Vphr), is an image of the right-hand side, noun_phrase (Nphr), verb_phrase (Vphr). This fact invites the possibility of writing an algorithm to generate these new structure arguments. This could be accomplished in the manner that the variables for difference lists get created and inserted into the underlying horn clause notation. We have experimented with such an algorithm and think it would be feasible. However, we do not include the implementation in this report.

For our present purposes we have a good reason for not automating the the specification of structures in this way. The structures one would like the theorem prover to return may not be isomorphic images of the surface phrase structure. Perhaps we would like the parse tree to display root forms of verbs or to reorder phrases. In section 4 we will present rules that perform these kinds of transformation. These transforms are easy to implement in Prolog and this ranks as what is perhaps the most important attribute of DCG's.

3.4 Terminal Rules

The grammar in section A.3 also deals with its dictionary in a different way. Words (terminal nodes) are now specified by defining various predicates like `is_determiner` and `is_noun`. And a set of terminal rules such as

`noun -> {is_noun}.`

is now responsible for making these dictionary entries accessible to the higher level rules.* Dictionaries now become somewhat easier to write and maintain.

3.5 Interfaces

We have created two modules to make using and building DCG's easier. These are PROSENT and PROTREE. One uses PROSENT to enter sentences which will be parsed by a grammar. PROTREE produces the graphical display of a parse tree. The Lisp codes for each module appear in appendix B and appendix C along with short explanations of how they work.

*Use of braces on the right side of a Prolog grammar rule turns off the automatic creation of a difference list. So things wrapped in braces are not taken to be constituents of the sentence or phrase.

If the two modules have been loaded into the Lisp environment, entering (*sentsys*) from a Lisp Listener window will cause a window to be created for entering sentences, as in figure 5.

To the prompt *Enter the sentence to parse ('Q' exits)*: appearing in that window, one can then enter a sentence as a sequence of words. Parentheses, quotes, periods, and other punctuation are not necessary and must not be used, unless the particular grammar rules in use recognize punctuation. The system then generates all possible parses of the sentence (as defined by the grammar), storing them in a global list. Finally, *sentsys* calls the function *draw-parse-tree*.

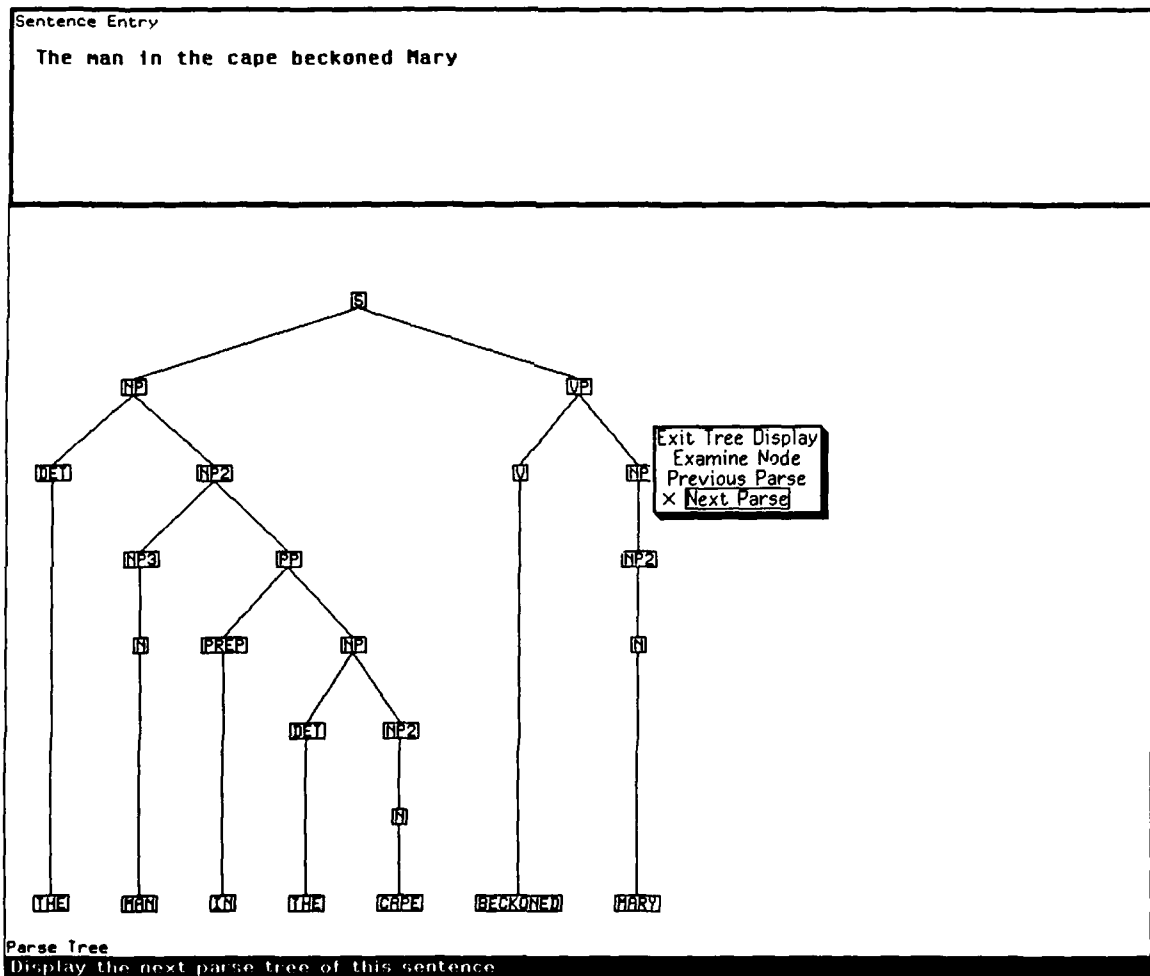


Figure 5: Mouse sensitive tree.

Draw-parse-tree creates a new window labeled *Parse Tree*, in which it draws the parse trees for the sentence. All terminal nodes, which usually represent actual words of the sentence, are dropped from their original positions in the tree to the bottom of the window, where they are all at the same horizontal level. This allows the words to be read directly across the bottom of the tree.

Each node of this graphical representation of the tree is sensitive to the mouse arrow, as shown in figure 5. With the mouse arrow over a node, clicking the right mouse button displays a menu with four options: *Examine Node*, *Next Parse*, *Previous Parse*, and *Exit Tree Display*. The first option opens a window which could be used to display further information about the node. This feature is not yet implemented. The second and third options are useful when more than one parse was generated by the grammar. The last option returns control to the sentence entry window.

4 Building a Grammar

4.1 Governing Considerations

As mentioned in the introduction, we are interested in parsers that are sensitive to the surface syntax of sentences. Our first implementation will look like an enhanced form of the tiny grammar in section A.3, and some of the enhancements will look like explicit statements that capture some of the well-known facts about English grammar. There are three general sorts of reason for proceeding in this way.

First, many of these rules of grammar are valid in the sense that they work. This means they more or less accurately describe the forms of actual speech and writing. People don't say things like

ate stallion apple the black

nor do they often say things like

these black stallion ate an apples.

Insofar as we can rely on adherence to correct grammar, we can use its rules to parse sentences. This means that we can significantly constrain the

rules to parse sentences. This means that we can significantly constrain the possible interpretations of a sentence. In fact, without relying on grammar in some way, it may be impossible to process a message in the form of a sentence.*

A reason for beginning with a syntactic parse is that its tree can be useful to the next (pragmatic/semantic) stage of processing. This second stage will generate some sort of logical form for the sentence about the stallion and the apple. Among other things, the fact that it is the stallion that is black and not the apple would be represented in this logical form. The parse trees generated by our rules of grammar advance the route to logical form by placing the modifiers of the noun off neighboring NP branches of the tree.

Third, in much of message and utterance understanding, the surface forms of a sentence carry special information about the speaker's (or sender's) meaning.[†] In our present example, use of the words *the black stallion* would normally convey the presupposition that a particular stallion is already in range (perhaps known to both speaker and hearer). By contrast, the words *the only black stallion in existence* do not convey this presupposition. However, some logicians would translate sentences using either choice of words into the same logical form; perhaps something like

$$(\exists x)(stallion(x) \wedge black(x) \wedge ate_an_apple(x) \wedge$$

$$(\forall y)(stallion(y) \Rightarrow x = y)).$$

Parsing rules that translate surface forms directly into this kind of logical form can obscure important distinctions.

4.2 General Constraints

The above considerations, especially the first (that the rules of grammar capture actual language use), will justify a fairly straightforward implementation of rules about number agreement, case, and person, as well as transitivity of verbs.

*See the comments in section 5, Ground not Covered, regarding processing bad grammar. Neither humans nor machines could process sentences without having internalized some kind of grammar. This means that even cases of bad grammar may really be cases of nonstandard grammar. If the sentence can be processed, then most likely there is some grammar into which the sentence fits.

[†]The surface form is just the literal appearance of a sentence as distinguished from any form that may result from processing the sentence. For example, *John hit Mary* and *Mary was hit by John* have different surface forms, although the two sentences probably have the same logical form.

4.2.1 Number Agreement

Verbs and their subjects, as well as most nouns and their determiners, must agree in number (also known as plurality). They must both be either singular or plural (the only two grammaticalized quantities in English). In a DCG, an obvious way to proceed is to first change the sentence rule

```
sentence(s(Nphr,Vphr) -> noun_phrase(Nphr),  
        verb_phrase(Vphr)).
```

to a rule with new arguments:

```
sentence(s(Nphr,Vphr)) -> noun_phrase(Nphr,Number),  
        verb_phrase(Vphr,Number).
```

Now only noun phrases and verb phrases with their **Number** arguments unified to the same value will satisfy the sentence rule. The sentence predicate needs no **Number** argument, because the concept of plurality does not apply to sentences. Of course, these values (for the variable **Number**) must be inherited ultimately from the Prolog unit clauses that make up the dictionary. So a full implementation of number agreement also requires modifications to all **noun_phrase** rules and **verb_phrase** rules as well as certain dictionary entries. Examples of some of what is needed are

```
noun_phrase(np(Det,Nphr2,Number)) ->  
determiner(Det,Number), noun_phrase_2(Nphr2,Number).
```

```
verb_phrase(vp(V),Number) -> verb(V,Number).
```

```
verb(v(V),Number) -> [V], {is_verb(V,Number)}.
```

```
is_determiner(a, singular).
```

```
is_noun(stallions, plural).
```

```
is_verb(ate, singular).
```

```
is_verb(ate, plural).
```

These kinds of change take care of most (but not all) of the constraints on number agreement among determiners, nouns, and verbs.

4.2.2 Pronoun Case

In English, case is grammaticalized only for pronouns, and only for three cases: nominative, objective, and possessive. The nominative case is used for two roles of the noun: subject of a sentence (or other finite clause) and predicate nominative. Examples are

I ate an apple

it was I.

The objective case is used for other roles such as direct object and indirect object.

Just as for number agreement, case can be handled by adding a new argument to certain predicates. However, this new case argument works to guarantee proper satisfaction of a clause based on position in the sentence rather than on an agreement between two parts of a sentence (as for number agreement).*

The rule for sentences becomes

sentence(s(Nphr,Vphr)) ->
noun_phrase(Nphr,Number,nominative),
verb_phrase(Vphr,Number).

The prepositional phrase rule becomes

prepositional_phrase(prphr(Prep,Nphr)) -> prepo-
sition(Prep), noun_phrase(Nphr,Number,objective).

Here we want constants rather than variables in the new argument positions. Again, the full implementation requires changing all the rules for noun phrase as well as some dictionary entries:

*In our DCG, role restrictions translate into position descriptions.

noun_phrase(np(Det,Nphr2),Number,Case))

->

determiner(Det,Number),

noun_phrase_2(Nphr2,Number,Case).

noun(n(N),Number,Case) -> [N], {is_noun(N,Number,Case)}.

is_noun(I,singular,nominative).

and so on.

4.2.3 Transitivity

Transitive verbs take direct objects; bitransitive verbs take both direct and indirect objects; and intransitive verbs must not have an object at all. Verbphrase rules that implement these restrictions could be written as

verb_phrase(vp(V,Nphr1),Number1) ->

verb(V,Number1,transitive),

noun_phrase(Nphr,Number2,objective).

verb_phrase(vp(V,Nphr1,Nphr2),Number1)

->

verb(V,Number1,bitransitive),

noun_phrase(Nphr1,Number2,objective),

noun_phrase(Nphr2,Number3,objective).

verb_phrase(vp(V),Number) -> verb(V,Number,intransitive).

In the above rule for bitransitive verbs, we must use three different **Number** variables, **Number1**, **Number2**, and **Number3**. The reason is that only the plurality of the verb determines the plurality of the parent verb phrase, **Number1**. The pluralities of the direct object and indirect object are not relevant; nor must their **Number** arguments be required to both instantiate to either singular or plural. To see this consider the two sentences

the man in the cape gave Mary the flowers

the men at the shore gave Mary a fish.

This completes our discussion of methods for implementing some of the general constraints of grammar. The remaining constraint, person, will be treated as part of the discussion of verbs.

4.3 Phrase Structures

Appendix D contains a collection of rules that incorporate all of the features urged so far. These rules also implement several, more elaborate forms of sentence. We believe that most of these rules can be understood directly by reading the listings in appendix D. Below we take up selected rules which may raise issues of particular interest.

4.3.1 Noun Phrases

The noun-phrase rules enforce distinctions among types of nouns. They provide separate treatment for pronouns (such as *I*, *they*, and *he*), common nouns (such as *river*, *stallion*, and *intelligence*), and proper nouns (such as *Mary*, *France*, and *Route 1*). Proper nouns and pronouns cannot ordinarily take adjectival modifiers or determiners. Equally important, these two kinds of noun function differently from common nouns in discourse. They can pick out or point to things without describing them. Common nouns, however, often provide information on classification and may be thought of as doing the work of adjectives.*

These facts and other related points become very important during the next (pragmatic/semantic) stage of processing which will operate on the parse tree. Hence, the parse tree must preserve these distinctions and mark the nodes accordingly, as shown in figure 6.[†]

Another type of noun phrase is the gerund phrase (see the third **noun-phrase** rule). Some rules for gerunds appear in section D.7. These rules can be used to parse sentences such as

crossing the river was dangerous

*Common nouns act like adjectives when used for describing something. When used for referring to something, their role changes. See Keith S. Donnellan [3].

[†]Beginning with figure 6, the parse trees are all generated by the grammar in appendix D. These trees contain nodes and configurations which have not yet been explained but will be in the sequel.

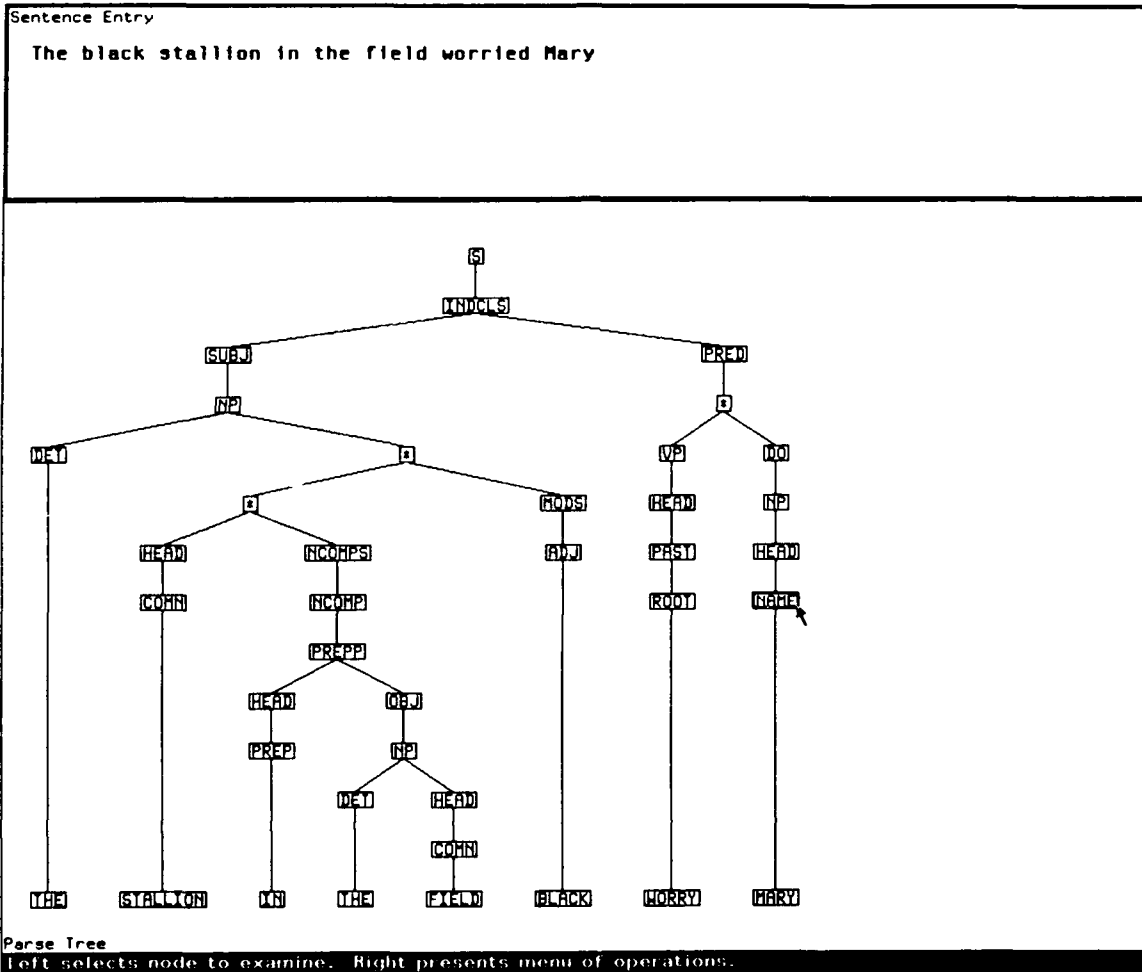


Figure 6: Noun phrases and higher structure.

and

his working here made everyone happy.

Gerunds are of some interest in the study of discourse about time and events. There is a very popular semantic theory of events (see Davidson [2]) which would translate (at some stage of processing) all action verbs like *went*, *win*, and *beckon* into nouns referring to events. This kind of translation can also be attempted at the level of surface structure, as in going from

Jack fell down the hill

to

Jack's falling was down the hill.

We therefore include gerunds for possible attention in the future.

Generally, common nouns can be modified by adjectives and determiners preceding and certain phrases, such as prepositional phrases, following. The rules for **noun_complement** in section D.8 could be expanded to cover more cases of the latter.

4.3.2 Higher Order Structure

The rules in section D.1 add new sentence parts to the grammar. The parse trees generated by these rules now display subjects, predicates, and independent clauses as in figure 6. And the noun phrases, verb phrases, and other phrases have nodes for a head, modifiers, and complements. One good reason for adding these parts is that the parse trees will now be easier to process in the next stage. A sentence may contain more than one noun and more than one verb. However, there is only one main verb and one main noun (barring conjunctions). *The more structured tree makes finding the main parts easier.*

In addition, the added structure makes writing good grammars easier. For example, the subject of a sentence need not be a noun phrase. Figure 7 illustrates an infinitive verb phrase as subject. The tree also explicitly marks this fact at the node INFVP.

4.3.3 Subordinate Adverb Clause

Other than the verb tenses (which we will discuss in sect. 4.3.5), perhaps the most important grammatical device for expressing temporal relations is the subordinate adverb clause. Our rule for this kind of subordinate clause is (see also sect. D.4)

```
subord_adv_clause(sadvcls(Subconj, Indcls))
->
subordconj(Subconj),
independent_clause(Indcls).
```

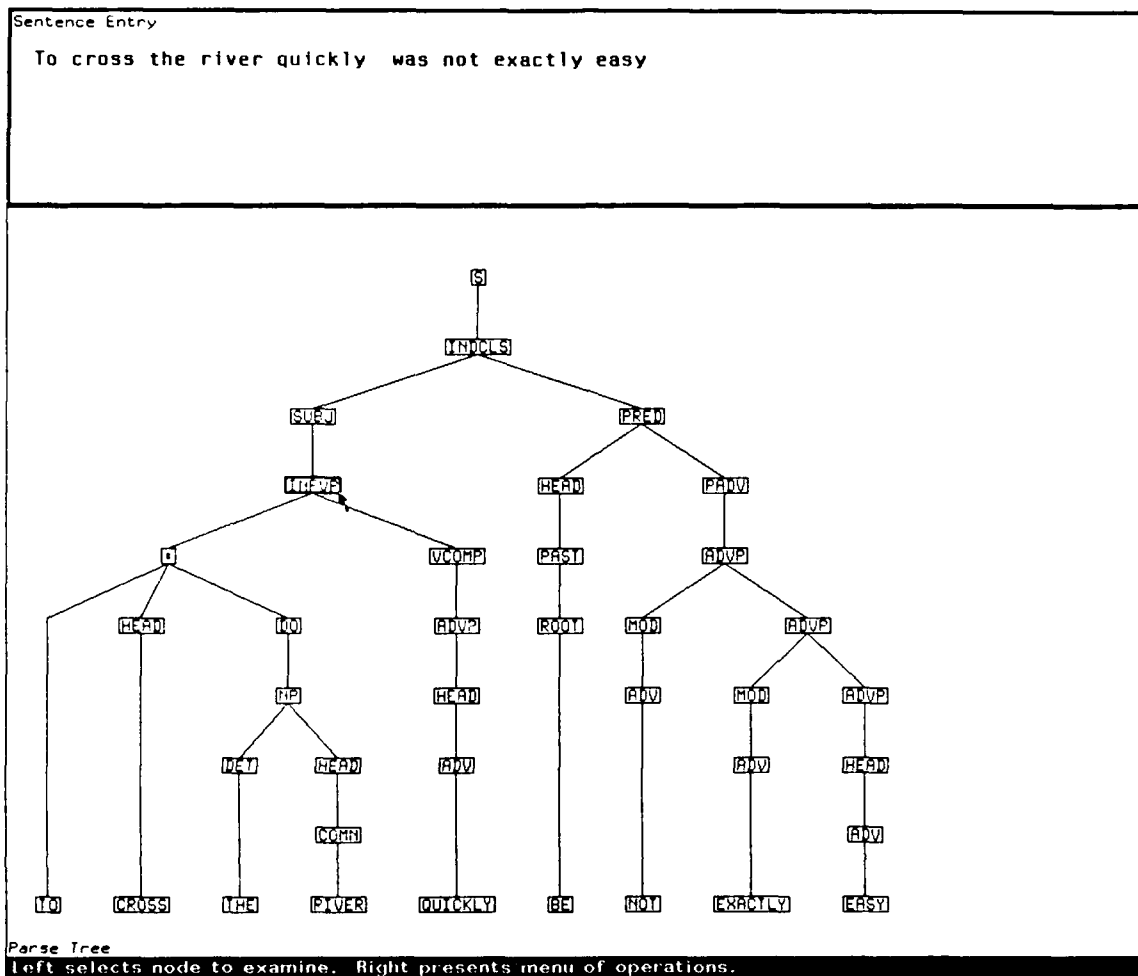


Figure 7: Infinitive verb phrase as subject.

This rule requires a new kind of dictionary entry for subordinating conjunctions, such as *before*, *after*, and *during* (see sect. D.12). It also makes use of the previously implemented top level structure for sentences, the independent clause. This choice means that our grammar will, in effect, accept any declarative sentence as the main part of a subordinate adverb clause. The grammar will not rule out the following, incoherent, sentence candidate,

I fed the horses after you will buy the oats,

because it does not enforce any tense agreement between main clause and subordinate clause. It is arguable that mistakes in tense agreement are prob-

lems for semantics and not for syntax. In our scheme, other stages of processing will be responsible for ordering events in time.

We incorporate the subordinate adverb clause into the grammar by defining a new type of sentence and a new type of independent clause (see sect. D.1). Figure 8 shows one example of the kind of parse tree generated.

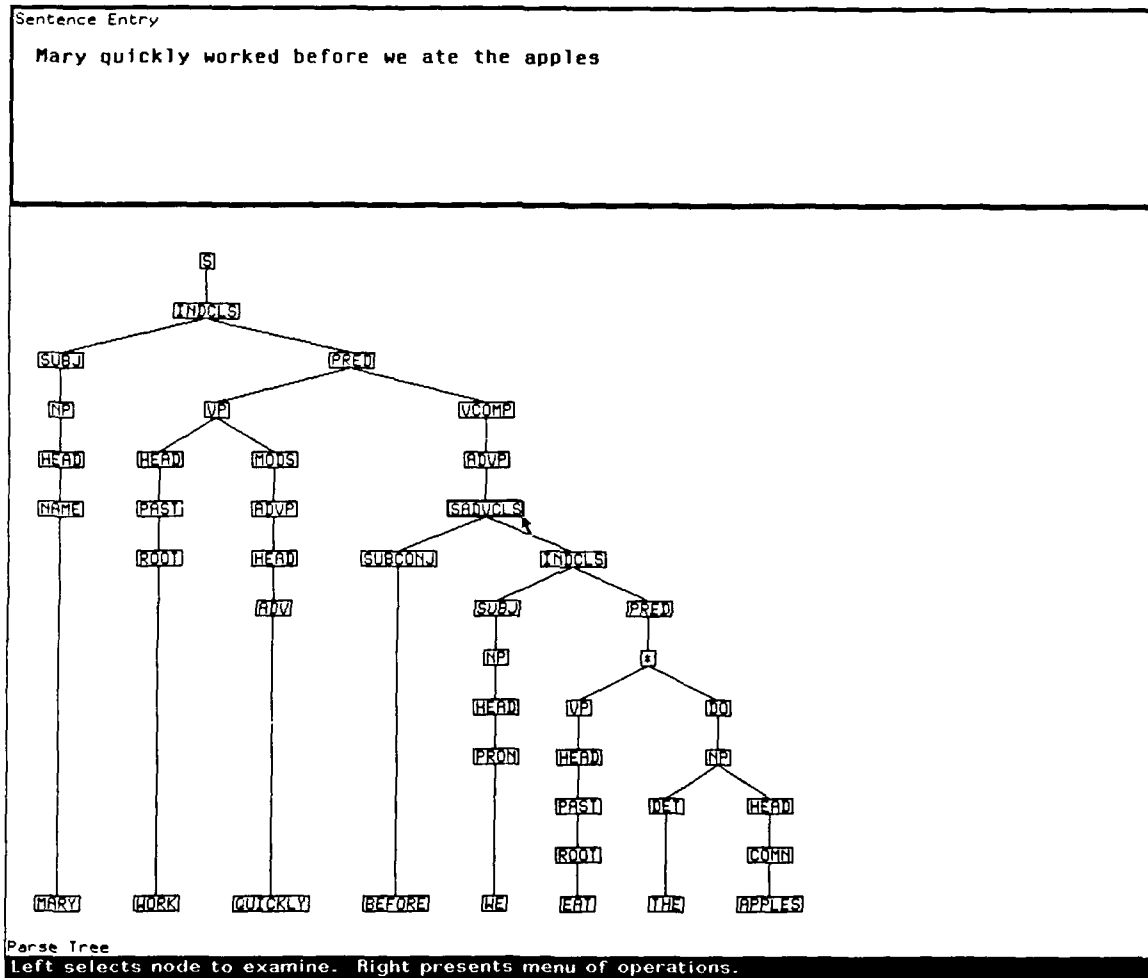


Figure 8: Subordinate adverb clause.

4.3.4 Sentence Prefixes

Now the second rule for independent clauses (sect. D.1),

```

independent_clause(indcls(Obj,
  pred(mods(rtshift(Advphr)),Vphr))) ->
  adverb_phrase(Advphr),
  subject(Obj, Number, Person),
  predicate(Vphr, Number, Person).

```

permits use of an adverb phrase preceding the subject of a sentence, as in

*before we crossed the river, the men worked near
Route 1.*

The parse tree appears in figure 9.

The structure argument in the above rule has been given special treatment in order to both facilitate the other stages of processing and preserve information about the original surface structure. This argument,

```
indcls(Obj,pred(mods(rtshift(Advphr)),Vphr))
```

is not isomorphic with the sequence of clauses on the right-hand side of the rule. The position of the adverb phrase has been moved to immediately precede the predicate, as is apparent in figure 9. This is necessary in order that the algorithms used in the next stage will be able to detect that the adverb phrase modifies the verb in the predicate and not the subject. Also, an extra node, called RTSHIFT, has been added to the tree. This signals the fact that the adverb phrase descending from this node has been moved inward. By these means, the grammar delivers a parse tree which is both easy to handle and full of information.*

4.3.5 Verb Tenses

The six tenses of the English verbs are past, present, future, and the perfect forms of each of these. Table 1 shows the tenses for the irregular verb *go*. Any system that will recover temporal information from discourse or text must parse the tenses correctly. Fortunately, a portion of the problem of

*This is just one example of the use of the structure argument in transforming surface structure into something more useful. Other examples will be presented in the section on verbs and in the later discussion of caveats and possible modifications to our collection of rules.

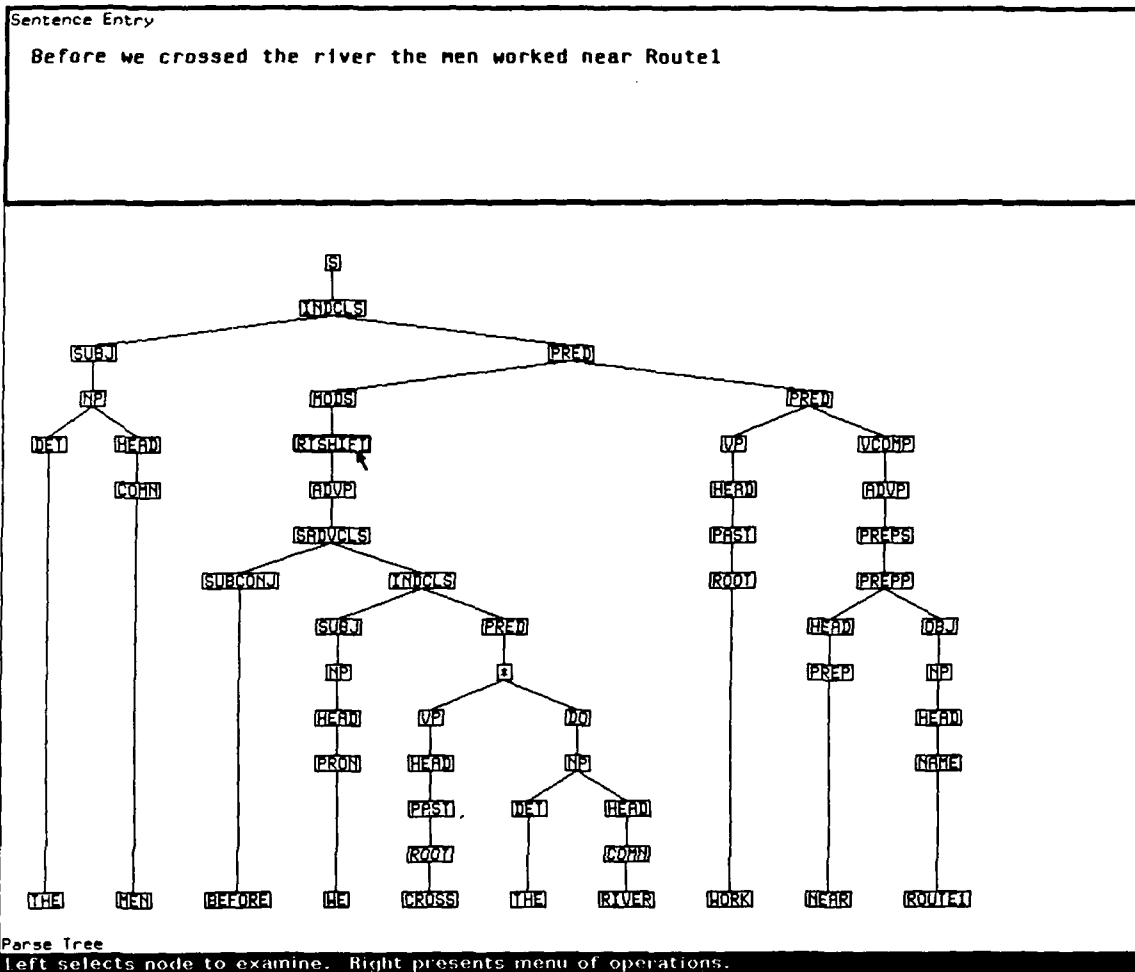


Figure 9: Leading adverb phrase.

Table 1: Verb Tenses

person	past	past perf	pres	pres perf	future	future perf
I	went	had gone	go	have gone	will go	will have gone
you	went	had gone	go	have gone	will go	will have gone
he	went	had gone	goes	has gone	will go	will have gone
they	went	had gone	go	have gone	will go	will have gone

tenses is fairly well bounded. Each tense for all action verbs uses the same auxiliary verbs (*had* for past perfect, etc.). And, except for the third person singular, all action verbs use the same conjugations for each tense: infinitive for present and future; past for past; and past participle for present perfect and future perfect. The only anomaly occurs for the third person singular form, which often requires a special conjugation for present tense and uses *has* in place of *have* for present perfect tense.

Our foundation for the higher level rules for verb tenses is the set of dictionary entries for action verbs (see sect. D.9). Here, each verb is entered with the predicate averb, which has six arguments. The pattern of arguments is

averb(Infinitive,Past,SingThird,PresPart,PastPart,Transitivity)

and the entry for the verb *go* is

averb(go,went,goes,going,gone,intransitive).

The rules for tenses in section D.9 access whichever of these six arguments is appropriate.

In all, there would be 42 rules for verb phrases covering the action verbs alone. One arrives at this number of rules through fairly strict adherence to declarative programming and a liberal interpretation of all the possibilities for combinations of main verbs, adverbs, temporal auxiliaries, and the special problem of the third person singular subject. We will take combinations of adverbs first.

No matter which tense we choose, we can think of a proper use of an adverb immediately preceding the kernel verb phrase. Cases include

i quickly went home

and

they easily will have finished the examination.

We save some redundancy in the writing of rules by making the first **verb_phrase** rule recursive (though not left recursive!):

```
verb_phrase(vp(Vphr,mods(Adv)),
  Number,Person,Type) ->
  adverb_phrase(Adv),
  verb_phrase_2(Vphr,Number,Person,Type).
```

Now we need only permute the legal combinations with adverbs between a temporal auxiliary and another verb. Such cases include

he had expertly groomed the stallion

and

they will already have come home.

As an example, one of the rules for future perfect is

```
verb_phrase_2(*(head(futperf(root(Infinitive))),
  mods(Adv)),Number,Person,Type) ->
  [will], [have],
  adverb_phrase (Adv),
  [V], {adverb (Infinitive,Past,SingThrd,PresPart,V,Type)}.
```

Note that this rule finds the correct conjugation by forcing the verb, which must unify with the clause [V], to unify with the argument for past participle in the dictionary entry. For future perfect there are three more rules to cover other permutations and combinations. And there are a few more for past perfect and present perfect.

Similar considerations for the singular third-person subject serve to further populate the set of present simple and present perfect rules. By using recursion and special tests, our final set (shown in sect. D.9), which covers all 42 cases, has been kept to 17 rules.*

The structure argument in the above future perfect rule,

*For a suggestion on how to begin building a recursive construction of tensed verb phrases as opposed to the somewhat tedious enumeration of cases employed here, see Pereira and Warren [5].

***(head(futperf(root(Infinitive))),mods(Adv))**

both manipulates and augments the original surface structure. This is apparent in the example shown in figure 10. The adverb has been moved to the rightmost position among the children of the node *. * This makes it easier to find. We have also replaced the surface verb, *gone*, with its infinitive, calling it the ROOT. This addition will facilitate semantic analysis because any information in the semantic knowledge base about going will all be stored under the entry GO. Finally, we have dropped the auxiliary verbs *will* and

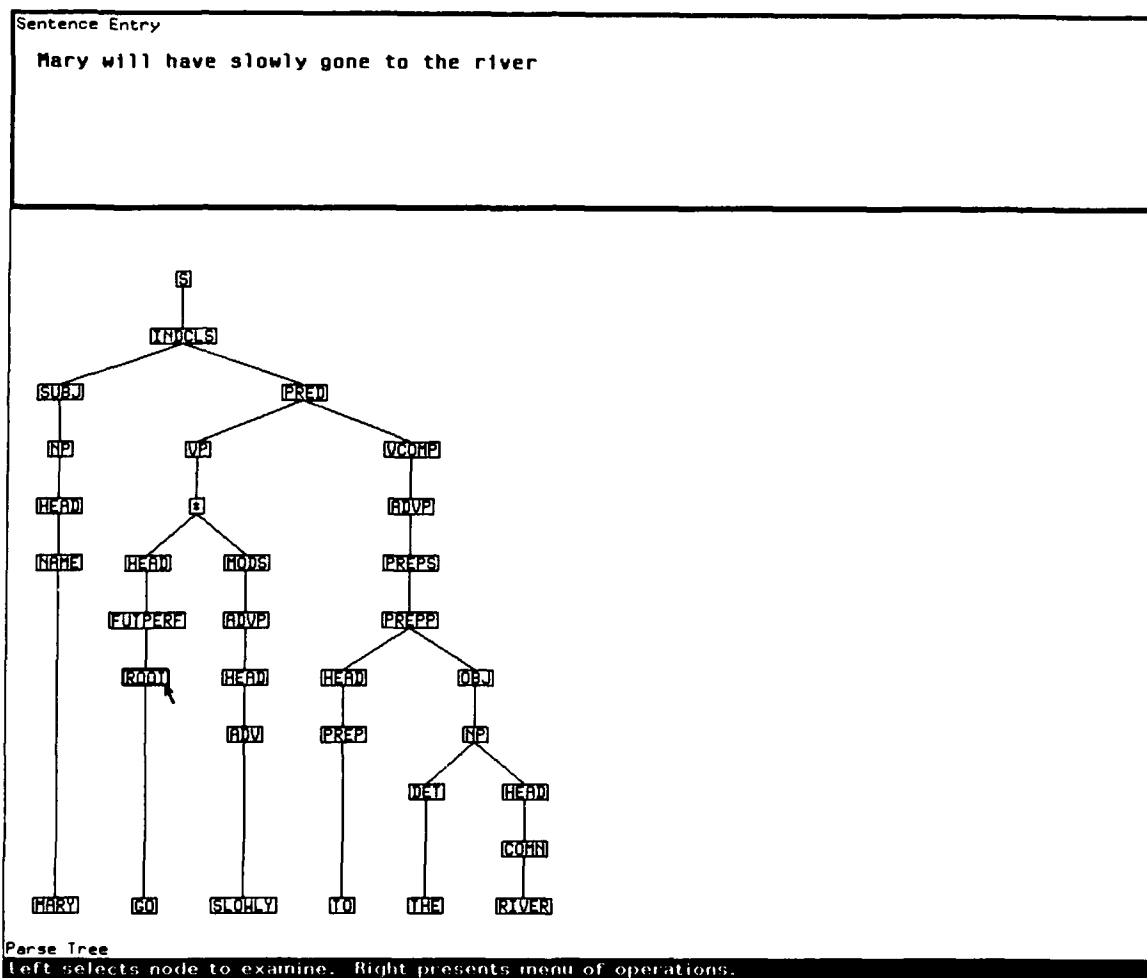


Figure 10: Tensed verb.

*Nodes labeled * are artifacts of the way the grammar rules are written. They can be thought of as being absorbed into their parents.

have, which do not appear in the parse tree. Apparently, the only function of temporal auxiliaries is to mark verb tense. This we do in the tree at the node FUTPERF. A simple traversal of the VP branch downward will recover and regiment the desired information about when and in what manner Mary approached the river.

A separate but similar set of rules for the verb *to be* is shown in section D.10. This verb is notoriously distinct in both syntax and meaning from the action verbs. Study of the rules for predicates (sect. D.3) will show some of the special syntactic behavior of this verb (which is treated as a special kind of sense verb *). Although *to be* participates in the usual six tenses, it is more irregular than the other verbs. Otherwise, the rules for *be_verb_phrase* resemble the rules for *verb_phrase*. In the dictionary (sect. D.12) we have entered the forms of the verb *to be* one by one using the predicate *beverb*. This is an alternative to the method that we used for action verbs, storing all verb forms under one predicate.

5 Ground Not Covered

Although our grammar largely accomplishes the goals laid down in the introduction, this should not be surprising. In fact, there are many structures, features, and restrictions of language that the grammar does not accommodate. It may be worthwhile to mention some of what is missing.

The sentence *Every man loves a woman* is ambiguous (though most readers will first think of only one of the two possible readings). At the level of logical form, the ambiguity can be explained as a matter of quantifier scope:

$$(\forall x)(man(x) \Rightarrow (\exists y)(woman(y) \wedge loves(x, y)))$$

versus

$$(\exists y)(\forall x)(man(x) \Rightarrow woman(y) \wedge loves(x, y)).$$

Our grammar gives no hint of this or any other problem about quantifiers. It would parse the sentence as a subject, *every man*, with a predicate consisting of a verb, *loves*, and a direct object, *a woman*.

It is certainly possible to hold that there is nothing wrong with this parse and that quantifier scoping should be handled by another module. The ques-

*Other verbs in this family include *seem*, *become*, and *look*.

tion is, in which module should we place the algorithms that handle quantifiers? Clearly, surface structure contributes relevant information here. But a good case can be made that in most discourse, other, pragmatic information must be used to (a) decide whether and what kinds of logical quantifiers to use and (b) determine their scopes. Further, in much dialog, quantifiers may not be called for.

Take an example involving the so-called anaphoric pronoun:

Jack took a number and divided it in two.

This seems to be a classic case of short-range syntactic anaphora. The pronoun *it* seems to stand for the noun *number*. On this interpretation, the logical form might resemble

$$(\exists x)(\text{number}(x) \wedge \text{took}(\text{jack}, x) \wedge \text{divided}(\text{jack}, x, 2)).$$

If this fact about *it* is totally a matter of syntax, shouldn't this recognition be built into the parser? The DCG could be rewritten to do the job.

But now consider the original sentence embedded in the following sequence:

*It was Jack's turn at the game of divide the pie.
His task was to pick a number from the jar and then
divide the pie into that number of pieces.
Jack took a number and divided it in two.
Which number did Jack pick?
Answer: the number two.*

Here again, the surrounding context of the sentence determines logical form.

Passives (*he was eaten by a lion*), imperatives (*go up on that ridge*), and questions (*where are they?*) cannot be parsed by our rules. Obvious forms of these structures could easily be added as new rules (making intelligent use of the structure argument).

Problems in dealing with conjunctions are notorious. Take a short example:

The dog bit my left hand and foot.

Does *left* modify *hand* only or should *left* be distributed over both *hand* and *foot*? Or should *left* modify the conjunctive phrase *hand and foot* as it would in the sentence

*the left hand and foot look like better specimens than
either those in the center or the single hand at
the far left.*

Of course, the syntactic parser cannot always make the proper determination unassisted. Perhaps it should deliver all three parses for later choice. Or perhaps it should deliver an ambiguous parse tree ready for further refinement. This seems to be what the current rules do about quantification.

Much talk about human activity uses the progressive aspect, as in

the group is crossing the river.

In our grammar, the predicate of this sentence gets parsed as the verb *to be* followed by a participial phrase. Thus, we do not treat the progressive as a verb type on a par with tensed verbs. We believe our treatment may be just as workable as other treatments which incorporate progressive verb forms into the rules of grammar.

Our grammar has leaks through which unacceptable sentences and unacceptable parse trees can enter. For example, our rules cannot show that *in front* modifies only the noun phrase and never the verb phrase in

the men in front will have crossed the river.

Other leaks involve various devices for negation. The rules do not disallow

they not will have crossed the river.

In DCG's, new rules, new argument places for predicates, and new conditions (wrapped in braces on the right-hand sides of rules) can all be used to add more restrictions to handle these cases.

It seems that, if desired, any of the above additional structures and restrictions could be added to the DCG. This suggests that the existing DCG

could undergo incremental growth to accommodate these things. Whether the incremental approach will always satisfy can probably be questioned. A less naive, more theoretically based, and more parsimonious grammar would gain greater syntactic coverage more elegantly (see, for example, McCord [4]). And any serious attempt to advance the state of the art in parsing English would be driven to such a grammar. The advantage of a naive grammar such as the one developed for this report is in ease of writing. For certain simple constructions and for an introduction to parsing, this grammar seems adequate.

Some real-time discourse is not grammatically correct. Noting this, one may be tempted to leave off using rules that seem to require exact conformity to correct syntax. Recalling the previous remarks in section 4.2, perhaps one should resist the temptation. On this matter it is relevant to point out that the concept of deviant syntax presupposes a concept of correct syntax. The fact of deviant syntax should make for more work in building parsers rather than suggest an escape from the approach taken in this report.

6 Concluding Comments

The work reported here represents an initial attempt at implementing a DCG in service of other studies in language processing. We have used the DCG to implement modularized syntactic parsing of a few forms of sentence. A goal for the future will be to look at what kinds of parse trees one might want the syntactic module to deliver. By making free use of the structure argument, we have found that DCG's are most well suited to this kind of study. DCG's are easy to write, easy to use, and easy to modify.

All of the advantages found in a DCG carry over to possible real-world applications. But real-world applications raise questions which we have not addressed. One sort of question centers around coverage; does the grammar handle any utterance or text it might really encounter? We have hinted at the vastness of natural language in various places above. Perhaps one can contrive applications where the coverage required can be limited in some reasonable, though still useful ways. Assuming this, questions of processing time may become important. How long will the system take to process each message, utterance, or piece of text?

In a research environment, the system takes too long if waiting for a parse to complete is annoying or if there is the worry that parsing time may

annoy an audience. In a real-world application, parsing time must (only) meet requirements of some larger system. One of the possible applications of the work described in the introduction is autonomous processing of streams of messages in background. In this case there is no immediate interaction with a human user and processing time may be less important. Where there is human interaction, perhaps something less than one second per sentence will do for an average parsing time.

DCG's implemented in Prolog are top-down parsers. Parsing then becomes a matter of many trials at predicate satisfaction and unification. There is a great deal of time-consuming backtracking. And, since our grammar is largely a declarative encoding of facts about language, there is little time saving control built into the grammar. The philosophy here was to concentrate on the logic of the problem at hand, leaving processing to the underlying implementation of Prolog. (The implementation on the Symbolics 3675 is fast and it uses indexing of predicates to achieve very fast searching for the right predicate to satisfy.)

Here are some parsing times for typical sentences using the grammar in appendix D:

before Mary worked they had eaten (30 ms)

before Mary will have worked they will have eaten (31 ms)

Mary will have worked before they will have eaten (274 ms)

Mary had it before they had it (174 ms)

Mary will have worked (2 ms)

Mary will have it (10 ms)

the black stallion ate an apple (6 ms)

The backtracking discernible in these results is caused by trying out false leads. It follows that the smaller the grammar, the shorter the parsing time. To illustrate, take *the black stallion ate an apple*. We used this sentence as the first example fed to the tiny grammar shown in section A.3. The

time that grammar takes to process the sentence is only 0.47 ms, an order of magnitude faster than the much larger grammar of appendix D. We conclude that parsing-time benchmarks can be misleading and perhaps useless in themselves.

Literature Cited

1. W.F. Clocksin and C.S. Mellish, *Programming in Prolog*, Springer-Verlag, Berlin, 1981.
2. Donald Davidson, *The Logical Form of Action Sentences*, in *The Logic of Decision and Action* (ed. by N. Rescher), University of Pittsburg, Pittsburg, 1967.
3. Keith S. Donnellan, *Reference and Definite Descriptions*, *Philosophical Review*, Vol. LXXV, 1966, pp. 281-304.
4. Michael C. McCord, *Using Slots and Modifiers in Logic Grammars for Natural Language*, *Artificial Intelligence*, Vol. 18, 1982, pp. 327-367.
5. Fernando C.N. Pereira and David H.D. Warren, *Definite Clause Grammars in Language Analysis*, *Artificial Intelligence*, Vol. 13, 1980, pp. 231-278.
6. Leon Sterling and Ehud Shapiro, *The Art of Prolog*, MIT, Cambridge, Mass., 1986.
7. *User's Guide to Symbolics Prolog*, Symbolics, Inc., Cambridge, Mass., 1986.
8. John E. Warriner, *English Grammar and Composition — Fourth Course*, Harcourt Brace Jovanovich, Orlando, 1982.
9. Terry Winograd, *Language as a Cognitive Process*, Volume 1: *Syntax*, Addison Wesley, Reading, 1983.

Appendix A Simple Grammars

Here are three simple but progressively better definite clause grammars.

A.1 A Simple Grammar

```
%%% -*- mode: prolog; syntax: prolog; package: prolog-user; -*-
%% A simple grammar, translated into prolog
sentence --> noun_phrase, verb_phrase.
noun_phrase --> determiner, noun_phrase.
noun_phrase --> noun.
noun_phrase --> adjective, noun_phrase.
noun_phrase --> noun_phrase, prepositional_phrase.
verb_phrase --> verb.
verb_phrase --> verb, noun_phrase.
verb_phrase --> verb, prepositional_phrase.
prepositional_phrase --> preposition, noun_phrase.
%% the dictionary
determiner --> [the].
determiner --> [a].
determiner --> [an].
noun --> [stallion].
noun --> [apple].
noun --> [man].
noun --> [men].
noun --> [mary].
noun --> [cape].
noun --> [river].
adjective --> [black].
verb --> [beckoned].
verb --> [ate].
verb --> [crossed]
preposition --> [in].
```


Appendix A

A.2 A Better Grammar

```
%%% -*- mode: prolog; syntax: prolog; package: prolog-user; -*-

%% A less simple grammar, translated into prolog
%% This grammar avoids infinite loops caused by left recursion
%% in grammar rules

sentence --> noun_phrase, verb_phrase.
noun_phrase --> determiner, noun_phrase_2.
noun_phrase --> noun_phrase_2.
noun_phrase_2 --> noun.
noun_phrase_2 --> adjective, noun_phrase_2.
noun_phrase_2 --> noun_phrase_3, prepositional_phrase.
noun_phrase_3 --> noun.
verb_phrase --> verb.
verb_phrase --> verb, noun_phrase.
verb_phrase --> verb, prepositional_phrase.
prepositional_phrase --> preposition, noun_phrase.
%% the dictionary
determiner --> [the].
determiner --> [a].
determiner --> [an].
noun --> [stallion].
noun --> [apple].
noun --> [man].
noun --> [mary].
noun --> [cape].
adjective --> [black].
verb --> [beckoned].
verb --> [ate].
preposition --> [in].
```

A.3 A Parsing Grammar

```

%%% -*- mode: prolog; syntax: prolog; package: prolog-user; -*-

%% A simple structure grammar, translated into prolog
%% The first argument in each rule stores the syntactic structure
%% which gets built up during parsing

sentence(s(NPhr,VPhr)) --> noun_phrase(NPhr), verb_phrase(VPhr).
noun_phrase(np(Det,NPhr2)) --> determiner(Det), noun_phrase_2(NPhr2).
noun_phrase(np(NPhr2)) --> noun_phrase_2(NPhr2).
noun_phrase_2(np2(N)) --> noun(N).
noun_phrase_2(np2(Adj,NPhr2)) --> adjective(Adj), noun_phrase_2(NPhr2).
noun_phrase_2(np2(NPhr3,PPhr)) --> noun_phrase_3(NPhr3), prepositional_phrase(PPhr).
noun_phrase_3(np3(N)) --> noun(N).
verb_phrase(vp(V)) --> verb(V).
verb_phrase(vp(V,NPhr)) --> verb(V), noun_phrase(NPhr).
verb_phrase(vp(V,PPhr)) --> verb(V), prepositional_phrase(PPhr).
prepositional_phrase(pp(Prep,NPhr)) --> preposition(Prep), noun_phrase(NPhr).

%% terminal rules
noun(n(N)) --> [N],{is_noun(N)}.
determiner(det(D)) --> [D],{is_determiner(D)}.
adjective(adj(A)) --> [A],{is_adjective(A)}.
verb(v(V)) --> [V],{is_verb(V)}.
preposition(preop(P)) --> [P],{is_preposition(P)}.

%% the dictionary
is_determiner(the).
is_determiner(a).
is_determiner(an).
is_noun(stallion).
is_noun(apple).
is_noun(man).
is_noun(men).
is_noun(mary).
is_noun(cape).
is_noun(river).
is_adjective(black).
is_verb(beckoned).
is_verb(ate).
is_verb(crossed).
is_preposition(in).

```

Appendix B PROSENT

Below is the listing for the module PROSENT. The top level function is *sentsys*.

The operation of this Lisp function *sentsys* (shown in the listing below) can be described as follows. The input sentence is read as a string and then converted to a Lisp list. This list is locally bound to the variable *sent*. To parse the sentence, *sentsys* must call Prolog. The macro *with-unbound-logic-variables* generates a Prolog variable, named *prs*. Next, the macro *with-query-satisfied* in effect submits the query (*sentence prs sent nil*) to Prolog. As each successful parse is returned (bound to *prs*), it is pushed on the global list **parse-list**. After parsing and receiving an appropriate reply from the user, *sentsys* calls *draw-parse-tree* (described in app. C).

```
;;; -*- Mode: LISP; Syntax: Zetalisp; Base: 10; Package: PL-USER; -*-

; present.lisp

(defvar *parse-list* nil "List of possible parse trees")
(defvar *sentence-symbol* 'sentence "So we are expecting to use a grammar with
'sentence' as the top level symbol for non-terminal node")
(defvar *sentwin* nil "This is the window for sentence entry.")
(setq *sentwin* nil)
(defun set-up-sentwin ()
  "Creates window for sentence entry."
  (setq *sentwin* (tv:make-window 'tv:window
    ':default-style '(:fix :bold :large)
    ':borders 3
    ':expose-p t
    ; make a wide short window at top of screen
    ':edges '(64 43 1131 200)
    ':label ""
    ':blinker-p t
    ':save-bits t)))

(defun sentsys ()
  "Gets a sentence and draws the parse tree"

  ;; check for existence of *sentwin*
  (cond (*sentwin* nil)
    (t (set-up-sentwin)))
  (let (phrase)
```

Appendix B

```
;; loop until the user is done entering sentences
(send *sentwin* :select)
(loop with done = nil
      until done do

  ;; output prompt to user
  (send *sentwin* :set-cursor-visibility :on)
  (send *sentwin* :clear-window)
  (scl:with-character-style ('(:fix :roman :normal) *sentwin*)
    (send *sentwin* :line-out " Enter the sentence to parse ('Q' exits): "))

;; Read a line of text. Then convert it into a lisp form with parens around it.
(scl:with-character-style ('(:fix :roman :normal) *sentwin*)
  (send *sentwin* :line-out ""))
(format *sentwin* " ")
(setq phrase (read-line *sentwin*))
(let ((sent (read-from-string (string-append "(" phrase ")") )))
  ;; Q exits
  (if (equal sent '(Q))
      (setq done t)

      (setq *parse-list* nil)

      ;; Get all possible parses of the sentence
      (with-unbound-logic-variables (prs)
        (with-query-satisfied '(*sentence-symbol* ,prs ,sent nil)
          (push (read-from-string (format nil "~A" prs)) *parse-list*)
          nil))

      (cond ((equal *parse-list* nil)
              (scl:with-character-style ('(:fix :roman :normal) *sentwin*)
                (send *sentwin* :line-out " "))
              (scl:with-character-style ('(:fix :roman :normal) *sentwin*)
                (send *sentwin* :line-out
                  " That is not a grammatical sentence.")))

            (t (loop with exit = nil
                      with parsenum = 0
                      with len = (1- (length *parse-list*))

                      do
                        (scl:with-character-style ('(:fix :roman :normal)
                                                      *sentwin*)
                          (send *sentwin* :line-out " ")))
```

```

(send *sentwin* :line-out
  (format nil
    " Hit a key to display tree ^A out of ^A . . ."
    (1+ parsenum)
    (1+ len))))
(send *sentwin* :tyi)
(send *sentwin* :clear-window)
(scl:with-character-style '(:fix :roman :normal)
  *sentwin*)
  (send *sentwin* :line-out "Sentence Entry")
  (send *sentwin* :line-out "")
  (format *sentwin* " "))
(send *sentwin* :set-cursor-visibility :off)
(send *sentwin* :line-out phrase)
(setq exit (draw-parse-tree (nth parsenum *parse-list*)))
  until (equal exit 'exit)
  if (equal exit 'next)
    do (setq parsenum (if (equal parsenum len)
                          0
                          (1+ parsenum)))
    else
      do (setq parsenum (if (equal parsenum 0)
                            len
                            (1- parsenum))))))

;; After the user is completely finished, bury the window
(send *sentwin* :bury)
(cond (*outwin*
      (send *outwin* :bury)))
'Done))

```

Appendix C PROTREE

The operation of PROTREE (shown in sect. C.2 below) can be described as follows. The function *draw-parse-tree* takes a parse tree list (as delivered by *sentsys*, described in app. B) as its argument and draws the corresponding tree. This process divides into five steps. First, the mouse-sensitive menus and output window (described in sect. 3.5 in the main body of the report) are set up. Second, the nested list form for the parse tree is converted into a one-dimensional array, in which each element is an instance of the Lisp flavor *tree-node*. Third, the proper display font is chosen. Fourth, the screen positions of the nodes are set. Finally, the nodes and arcs (straight lines) are drawn.

The function *make-mouse-menu* sets up the menus. And the function *init-window-io* creates the window **outwin** as an instance of the flavor *treewindow* to complete the first step.

The second step is accomplished by the function *build-array* which is called with three arguments: the array to receive the parse tree, the list form of the tree, and the parent of the current node to be entered into the array. The parent of the top node is *nil*. *Build-array* first makes an instance of the flavor *tree-node* to represent the current node. This flavor has five instance variables: *parent*, which contains the array position of the parent of the current node; *children*, which contains a list of array positions of any children; *label*, which contains the text to be displayed at this node; and *x* and *y*, which contain the screen coordinates of the top edge of the node. This flavor may be expanded in the future to store other information, such as the root form of a verb, whether the node has been moved to a new position by a transformation rule, or the synonym for a word.

The two instance variables *parent* and *label** are set immediately by *build-array*. Then, if the node has children, *build-array* invokes the flavor method *set-children* to both set the value of *children* and call *build-array* recursively on each child.

After traversing the parse tree, *build-array* calls *drop-terms* in order to move all terminal nodes down to the same horizontal position. *Drop-terms* inserts extra nodes, each with the label "|" between terminal nodes and penultimate nodes as required. Thus, these nodes look like segments of the

*Currently, the label is taken as the head of the list which delimits a node in the parse tree (e.g., subj, np2, stallion).

Appendix C

arcs which will appear between nodes.

The function *get-font* is called to accomplish the third step in *draw-parse-tree*.*

In the fourth step, the function *set-terminal-coordinates* spaces the terminal nodes equally along the bottom of the screen. Next the function *set-x-coordinates* recursively moves up the tree (from terminals to top node) by placing nonterminal nodes either directly over single children or mid-way between two or more children. Finally, *set-y-coordinates* moves up the tree while incrementing vertical positions of nodes at each level.

The tree is drawn by sending the top node the message *:draw*. This flavor method gets the *x* and *y* coordinates and the label from the node's flavor-instance, which is now fully instantiated. If the node is a padding node, with label "|", a vertical line is drawn. Otherwise the node's label is drawn and enclosed in a box. A mouse-sensitive area is inscribed over this box. Recursive calls to *:draw* repeat these operations for the remaining nodes in the tree.

C.1 Suggested Modifications to PROTREE

Currently the terminal nodes are spaced widely to ensure that none of the nonterminals will overlap on the screen. A more elegant solution would be to scan the tree breadth first to find nodes that would overlap with their left siblings. These nodes would then be moved rightward with parents and children following. The tree must then be rescanned as before.

Parse trees for sentences which are either much longer or more complex than those shown in figures 9 and 10 will not fit in our window. The obvious solution is to employ some variety of window scrolling so that portions of the tree can be displayed in succession. This could take the form of redrawing portions of the tree, or use of scrolling windows, or use of the Symbolics Genera 7.1 presentation system.

*The listing for PROTREE shown in this report makes font choosing ineffectual because there is but one font on the list of fonts. If there were other fonts, *get-font* would choose the largest of these possible for fitting the terminal nodes across the screen.

C.2 Draw-Parse-Tree and Supporting Functions

```

;;; -*- Package: PL-USER; Mode: LISP; Base: 10; Syntax: Zetalisp -*-

;; Flavor for a pointed tree node
(defflavor tree-node
  ( (parent nil) ; number of parent node
    (children nil) ; list of child node numbers
    x ; xcoor for display on screen
    y ; ycoor for display on screen
    (label "") ; text label of node
    ) ()
  :settable-instance-variables)

(zl:defconst *outwin* nil) ; output window for parse tree
(zl:defconst *xlim* 900) ; maximum x limit for tree size
(zl:defconst *ylim* 543) ; " " " y " " " " " "
(zl:defconst *maxheight* 550) ; maximum height of tree
(zl:defconst *vertgap* 4) ; minimum vertical gap between tree levels,
                          ; measured in character lines

;; the list of fonts used by the program
;; format is (style baseline), where baseline was taken from the font attributes
(zl:defconst *fontlist* '(((fix :roman :normal) 10)))

;; ((dutch :roman :very-small) 8)))
; degenerate list of fonts

;; For auto choosing of font size set *fontlist* to following

; '(((fix :roman :very-large) 16)
; ((dutch :roman :normal) 13)
; ((fix :roman :normal) 10)
; ((dutch :roman :very-small) 8)
; ((fix :uppercase :very-small) 5)))

;; flavor of a tree window. Uses mouse-sensitive items.

(defflavor treewindow ()
  (tv:basic-mouse-sensitive-items
   tv:window))

(zl:defconst thelist () ; list for accumulating mouse menu items

```


Appendix C

```
(zl:defconst *chheight* nil) ; character height

(defun build-array (trarray trlist parent)
  "Creates a pointed tree array out of the lisp list form of tree."

  (let ((node (make-instance 'tree-node
    ; create a new tree-node, labeled with the label of the list form
    :parent parent
    :children nil
    :label (string (if (listp trlist) (car trlist)
      trlist)) )))
    (let ((nodenumber (zl:array-push-extend trarray node)))
      ; add it to the tree

      (if (listp trlist) ; if there are children, create them
        (send node :set-children
          (mapcar 'build-array (circular-list trarray) (cdr trlist)
            (circular-list nodenumber))))

      (drop-terms 0 trarray (1- (depth 0 trarray)))
      ; drop the terminals to an even level for easy sentence reading

      ;; the next line returns the value of nodenumber
      nodenumber)))

(defun depth (nodeno tree)
  "Returns the depth of tree from nodeno down."

  (let ((node (aref tree nodeno)))
    (if (null (send node :children))
      1
      (1+ (apply 'max (mapcar 'depth (send node :children)
        (circular-list tree)))))))

(defun set-all-coords (tree style)
  "Sets all coords of all nodes of tree, with printing in style."

  (set-terminal-coords tree style)
  (set-node-coords tree))

(defun set-terminal-coords (tree style)
  "Set the coords of all terminals of tree, with printing done in style."

  (let ((terms (get-all-terms tree 0))) ; all the terminal numbers
```

```

(loop for terminal in terms
  for termnode = (aref tree terminal)

    ;; get the length in pixels of the label string
  for termilen = (zl:multiple-value ()
    (send *outwin* :string-length
      (format nil " ~A "
        (send termnode :label))
        0 nil nil style))

    for halflen = (zl:// termilen 2)
  for xcoor from 10 do ; current xcoor (updated for each terminal)
    ; set the x to xcoor modified by the string-length
    (send termnode :set-x (+ xcoor halflen))
    (setq xcoor (+ xcoor termilen 20)) ; reset xcoor
    (send termnode :set-y *ylim*))) ; y is set to the lowest level possible

(defun get-all-terms (tree nodenum)
  "Returns a list of terminal-node numbers of a tree from nodenum down."

  (let ((node (aref tree nodenum)))
    (let ((kids (send node :children)))
      (if (null kids)
        ; if it's a terminal, it's the only terminal returned
        (list nodenum)
        ; a list combining the terminals of its kids
        (apply 'append (mapcar 'get-all-terms (circular-list tree) kids))))))

(defun set-node-coords (tree)
  "Sets the coords of the tree nodes, assuming the terminals are already set."

  (set-x-coords 0 tree)
  (set-y-coords 0 tree))

(defun set-x-coords (nodenum tree)
  "Sets the x-coords of the tree starting at nodenum. Terminals are assumed to
  be set already."

  (let ((node (aref tree nodenum)))
    (let ((kids (send node :children)))
      (if (null kids)
        (send node :x) ; if it's a terminal, the coord is already set
        (let ((firstx 9999)
              (lastx 9999))
          (setq firstx (set-x-coords (car kids) tree))
          ; the first x-coor of the child nodes
          (setq lastx (set-x-coords (cadr kids) tree))
          (set-x-coords (+ firstx lastx) node))))))

```

Appendix C

```

(loop for child in (cdr kids) do
  (setq lastx (set-x-coords child tree)))
  ; the last x-coor of the child nodes
; if only one child, set this node to the child's x
  (send node :set-x (if (= lastx 9999)
    firstx
; otherwise the middle of the first and last kids x's
    (zl:// (+ firstx lastx) 2)
    )))))))

(defun set-y-coords (nodenum tree)
  "Sets the y-coords of the tree starting at nodenum. Terminals are assumed to
  be set already."

  (let ((node (aref tree nodenum)))
    (let ((kids (send node :children)))
      (if (null kids)
        (send node :y) ; if it's a terminal, the coord is already set
        (loop for child in kids
          for kid-y = 9999 do
            (setq kid-y (set-y-coords child tree))
            ; set the y-coords of the next level down
; set this y-coord to a level above its kids' ycoords
            finally (return (send node :set-y (- kid-y *vertgap*)))
            )))))

(defun drop-terms (nodenum tree drop)
  "Drops all terminals of a given node to the same depth by inserting nodes
  labeled '|'"

  (setq drop (1- drop)) ; decrease drop distance
  (let ((node (aref tree nodenum)))
    (let ((kids (send node :children)))
      (loop for child in kids ; for each child
        for chnode = (aref tree child) do

        ;; if the child is a terminal, drop it

        (if (null (send chnode :children))
          (if (> drop 0)
            (let ((tempnode node)
                  ; temporary variables for the current node and child
                  (tempkid chnode)
                  (tempnum nodenum)
                  (kidnum child))

```

```

        (loop for count from 1 to drop by 1
              ; for each level to drop node
; create a tree-node labeled "|"
              for newnode = (make-instance 'tree-node
                                           :parent tempnode
                                           :children tempkid
                                           :label "|")

; add to array, lengthen if necessary
              for newnum = (zl:array-push-extend tree newnode) do

                ;; insert newnode between tempnode and tempkid
                (send tempnode :set-children
; replace child with newnode
                  (subst newnum kidnum (send tempnode :children)))
; point the newnode to the tempkid
                (send newnode :set-children (list kidnum))
                (send newnode :set-parent tempnum)
                ; point the newnode up to tempnode
                (send tempkid :set-parent newnum)
                ; point the tempkid up to newnode
                (setq tempnode newnode)
                ;move the temp. var's down a level to drop again
                (setq tempnum newnum))
            )
        )
    ;; if the child is not a terminal, drop its terminals

    (drop-terms child tree drop)
  )
)
)
)
)

(defmethod (:draw tree-node) (tree style)
  "Draws the tree by 1. Drawing this node 2. Calling :draw-children"

  (let ((text (send self :label)))
    (let ((x2 (1+ (- (send self :x)
                     (zl://
                      (zl:multiple-value ()
; length in pixels of label
                        (send *outwin*
                           :string-length text
                           0 nil nil style))

```

Appendix C

```

                2))))
      (y2 (+ (send self :y) *chheight*)))

      (if (not (equal text "|"))                ; it's not a padding node
          (let ((x3 (+ (- (* 2 (send self :x)) x2) 3))
                (y3 (1- (send self :y))))
              (scl:with-character-style (style *outwin*)
; draws the label
                                (graphics:draw-string text x2 y2
                                :stream *outwin*))
              (let ((x2 (- x2 2))
                    (y2 (1+ y2)))
; draws the box
                    (send *outwin* :draw-lines tv:alu-ior x2 y2 x2 y3 x3 y3 x3
                    y2 x2 y2)
; sets up mouseable area
                    (send *outwin* :primitive-item ':node-type self x2 y3 x3 y2)))
          (let ((x3 (send self :x))
                (y3 (send self :y)))
; draws the vertical line for a padding node
                (send *outwin* :draw-line x3 y3 x3 (+ y3 *chheight*)))
          (send self :draw-children tree style))))

(defmethod (:draw-children tree-node) (tree style)
  "Draws a tree's arcs and send the children to :draw"

  (let ((len (length (send self :children)))
        (selfx (send self :x))
        (selfy (send self :y)))

    (if (not (zerop len))
        (loop for childno in (send self :children)
              for child = (aref tree childno)
              for xcoor = (send child :x)
              for ycoor = (send child :y) do
; draw the child's arc
              (send *outwin* :draw-line selfx (+ 1 selfy *chheight*) xcoor
              (1- ycoor))
              (send child :draw tree style))))))

(defun get-font (tree)
  "Choose the largest font possible for drawing the tree by checking to see if
  the terminals fit across the bottom"

  (loop for fontpair in *fontlist*
```

```

for style = (car fontpair)
with terms = (mapcar 'send
                    (mapcar 'aref (circular-list tree) (get-all-terms
                                                              tree 0))
; this double-mapcar returns a list of terminal labels
                    (circular-list ':label))

;; the next part gets the length in pixels of the string, in font "style"
for treewidth = (+ (* (length terms)
                      (zl:multiple-value ()
                        (send *outwin* :string-length (format nil " ")
                          0 nil nil style))))
                (zl:multiple-value ()
                  (send *outwin* :string-length
                        (format nil "~A" terms)
                          0 nil nil style))))

for dep = (depth 0 tree)
for treeheight = (* 4 (cadr fontpair) dep)
; it "fits" vertically if there is
; a gap of at least four rows between levels

until (and (< treewidth *xlim*) (< treeheight *maxheight*))
finally (setq *vertgap* (zl:// *maxheight* dep))
; the maximum gap which will still fit
finally (return fontpair)))

(defun make-mouse-menu ()
  "Creates the list to be used as a menu for mouse-sensitive areas"

  (tv:add-typeout-item-type thelist
    :node-type "Next Parse" next
    nil "Display the next parse tree of this sentence")
  (tv:add-typeout-item-type thelist
    :node-type "Previous Parse" prev
    nil
    "Display the previous parse tree of this sentence")
  (tv:add-typeout-item-type thelist
    :node-type "Examine Node" (examine-node)
    t "Examine details of node information")
  (tv:add-typeout-item-type thelist
    :node-type "Exit Tree Display" exit
    nil
    "Exit parse tree display and return to previous window")

```

Appendix C

```
(tv:add-typeout-item-type thelist
  :exit-type "Next Parse" next
  nil
  "Display the next possible parse tree of this sentence")
(tv:add-typeout-item-type thelist
  :exit-type "Previous Parse" prev
  nil
  "Display the previous parse tree of this sentence")
(tv:add-typeout-item-type thelist
  :exit-type "Exit Tree Display" exit
  t
  "Exit parse tree display and return to previous window")

(tv:add-typeout-item-type thelist
  :next-type "Next Parse" next
  t
  "Display the next possible parse tree of this sentence")
(tv:add-typeout-item-type thelist
  :next-type "Previous Parse" prev
  nil
  "Display the previous parse tree of this sentence")
(tv:add-typeout-item-type thelist
  :next-type "Exit Tree Display" exit
  nil
  "Exit parse tree display and return to previous window")

(tv:add-typeout-item-type thelist
  :prev-type "Next Parse" next
  nil
  "Display the next possible parse tree of this sentence")
(tv:add-typeout-item-type thelist
  :prev-type "Previous Parse" prev
  t
  "Display the previous parse tree of this sentence")
(tv:add-typeout-item-type thelist
  :prev-type "Exit Tree Display" exit
  nil
  "Exit parse tree display and return to previous window"))

(defun examine-node (node)
  "Examines node clicked on. Called by process-mouse. Not yet supported."

  (let ((examwin (tv:make-window 'tv:window
    ':edges '(64 43 1131 200)
    ':label "Examine Node"
```

```

                                ':expose-p t)))
  (graphics:draw-string
    (format nil "Node: ~A" (send node :label)) 100 25 :stream examwin)
  (graphics:draw-string
    "Examining nodes is not yet supported." 100 50 :stream examwin)
  (graphics:draw-string
    "Please hit a key to continue." 100 75 :stream examwin)
  (send examwin :tyi)
  (send examwin :kill)))

(defmethod (:who-line-documentation-string treewindow) ()
  "Left selects node to examine. Right presents menu of operations.")

(defun process-mouse ()
  "Gets mouse blip and executes matching function until exit"

  ;; turn off screen graying
  (tv:set-screen-deexposed-gray nil)

  ;; create "Exit" box in upper left corner
  ; (send *outwin* :primitive-item ':exit-type 'exit 54 40 225 59)
  ; (send *outwin* :draw-lines tv:alu-ior 52 38 52 59 225 59 225 38 52 38)
  ; (scl:with-character-style '(:fix :roman :very-large) *outwin*)
  ; (graphics:draw-string "Exit" 116 56 :stream *outwin*))

  ;; create "Next Parse" node in upper left corner
  ; (send *outwin* :primitive-item ':next-type 'next 54 62 225 81)
  ; (send *outwin* :draw-lines tv:alu-ior 52 60 52 81 225 81 225 60 52 60)
  ; (scl:with-character-style '(:fix :roman :very-large) *outwin*)
  ; (graphics:draw-string "Next Parse" 79 78 :stream *outwin*))

  ;; create "Previous Parse" node in upper left corner
  ; (send *outwin* :primitive-item ':prev-type 'prev 54 84 225 103)
  ; (send *outwin* :draw-lines tv:alu-ior 52 82 52 103 225 103 225 82 52 82)
  ; (scl:with-character-style '(:fix :roman :very-large) *outwin*)
  ; (graphics:draw-string "Previous Parse" 55 100 :stream *outwin*))

  ;; process mouse blips
  (loop for blip = (send *outwin* ':list-tyi)
        until (not (listp (cadr blip)))
        if (listp (cadr blip))
          do (eval (append (cadr blip) (list (third blip))))
        finally (send *outwin* :clear-window)
        finally (send *sentwin* :select)
        finally (tv:set-screen-deexposed-gray tv:6%-gray))

```


Appendix C

```
        finally (return (cadr blip))))

(defun draw-parse-tree (tree)
  "Draws the parse tree as returned by parse and allows examination of nodes"

  (make-mouse-menu) ; prepares mousable menu

  ;; test for existence of *outwin*
  (cond (*outwin*
        (send *outwin* :select))
        (t (init-window-io))) ; prepares output window

  (let ((*tree* (init-tree)) ; creates array of nodes to hold tree
        (*chheight* 0)) ; font baseline
    (build-array *tree* tree nil) ; fill the array with a tree of nodes/pointers
    (let ((fontpair (get-font *tree*))) ; (style font-baseline)
      (setq *chheight* (cadr fontpair))
      (set-all-coords *tree* (car fontpair))
      ; arrange node coords based on chosen font
      (send (aref *tree* 0) :draw *tree* (car fontpair))))
    ; draw the tree at chosen coords
    (process-mouse)) ; use the mouse to examine the tree.
                    ; Return 'next or 'exit

(defun init-window-io ()
  "Creates a window for displaying the parse tree"
  (setq *outwin* (tv:make-window 'treewindow
                                ':edges '(64 200 1131 793) ; 64 43 1131 793
                                ':label "Parse Tree"
                                ':borders 1
                                ':blinker-p nil
                                ':save-bits t
                                ':expose-p t
                                ':item-type-alist thelist
                                ; accumulates mouse menu items
                                ))

  (send *outwin* :select))

(defun init-tree ()
  "Creates an extendable array to hold tree-nodes"
  (make-array 10 :fill-pointer 0))
```

Appendix D The Grammar

Following are the listings of the rules and lexicon of the grammar introduced in section 4.3 of the main body of the report. The sections on the following pages divide this grammar into convenient bundles.

Appendix D

D.1 Sentences and Independent Clauses

```
%-----
% sentences
%-----

% independent clauses are sentences

sentence(s(Sent)) -->
    independent_clause(Sent).

% if/then statements are sentences

sentence(implies(Sent1, Sent2)) -->
    [if], independent_clause(Sent1),
    [then], independent_clause(Sent2).

sentence(implies(Sent1, Sent2)) -->
    [if], independent_clause(Sent1),
    independent_clause(Sent2).

%-----
% independent clauses
%-----

% canonical independent clause

independent_clause (indcls(Subj, VPhr)) -->
    subject (Subj, Number, Person),
    predicate(VPhr, Number, Person).

% adverb prefix to a sentence

independent_clause (indcls(Subj, pred(mods(rtshift(Advphr))), VPhr))) -->
    adverb_phrase (Advphr),
    subject (Subj, Number, Person),
    predicate (VPhr, Number, Person).

% independent_clauses using expletive "There" as empty subject ["There are apples"]

independent_clause (exists(NPhr)) -->
    [there, is],
```

Appendix D

```
subject (NPhr, singular, Person).  
  
independent_clause (exists(NPhr)) -->  
[there, are],  
subject (NPhr, plural, Person).
```

Appendix D

D.2 Subjects and Objects

```
%-----  
%% subject of a sentence  
%-----  
  
% a nominative case noun phrase is a subject  
  
subject(subj(NPhr), Number, Person) -->  
  noun_phrase (NPhr, Number, Person, nominative).  
  
% an infinitive verb phrase: "to run" is a subject  
  
subject (subj(IVP), singular, third) -->  
  inf_verb_phrase (IVP).  
  
%-----  
% other noun type parts  
%-----  
  
% a nominative case noun phrase is a predicate nominative  
  
pred_nominative (pdnom(NPhr), Number, Person) -->  
  noun_phrase (NPhr, Number, Person, nominative).  
  
% any adjective phrase is a predicate adjective  
  
pred_adjective (pdadj(Adj)) -->  
  adjective_phrase (Adj).  
  
direct_object (do(NPhr), Number, Person) -->  
  noun_phrase (NPhr, Number, Person, objective).  
  
indirect_object (io(NPhr), Number, Person) -->  
  noun_phrase (NPhr, Number, Person, objective).
```

D.3 Predicates

```

%-----
%% predicates
%-----

predicate(pred(Pred2), Number, Person) -->
  predicate_2(Pred2, Number, Person).

% verb phrase, prepositions
% example: [I nibbled the carrot in the garden.]

predicate(pred(Pred2, vcomp(Adv)), Number, Person) -->
  predicate_2(Pred2, Number, Person),
  adverbs(Adv).

% sense-verb -\- prepositional phrase
% example: [I am in the garden.]

predicate (pred(VPhr, padv(Advphr)), Number, Person) -->
  sense_verb_phrase(VPhr, Number, Person),
  adverb_phrase(Advphr).

% an intransitive verb cannot have a direct object

predicate_2 (VPhr, Number, Person) -->
  verb_phrase(VPhr, Number, Person, intransitive).

predicate_2 (Pred3, Number, Person) -->
  predicate_3(Pred3, Number, Person).

% sense verb -\- predicate nominative
% example: [I am a rabbit.]

predicate_2 (pred(VPhr, pnom(PredNom)), Number, Person) -->
  sense_verb_phrase (VPhr, Number, Person),
  pred_nominative(PredNom, Number, Person_2).

% sense verb -\- predicate adjective
% example: [I am angry.]

predicate_2(pred(VPhr, padj(Adj)), Number, Person) -->
  sense_verb_phrase(VPhr, Number, Person),

```

Appendix D

```
pred_adjective(Adj).  
  
% verb -\- direct object  
  
predicate_3 (*(VPhr, DirObj), Number, Person) -->  
  verb_phrase(VPhr, Number, Person, transitive),  
  direct_object(DirObj, Number2, Person2).  
  
% verb -\- indirect object -\- direct object  
  
predicate_3 (*(VPhr, IndObj, DirObj), Number, Person) -->  
  verb_phrase (VPhr, Number, Person, bitransitive),  
  indirect_object(IndObj, Number2, Person2),  
  direct_object (DirObj, Number3, Person3).
```

D.4 The Subordinate Adverb Clause

```
%-----  
% subordinate adverb clause  
%-----  
  
subord_adv_clause(sadvcls(Subconj, IndCls)) -->  
  subordconj(Subconj),  
  independent_clause (IndCls).
```


Appendix D

D.5 Infinitives

```
%-----  
%% infinitive verb phrases  
%-----  
  
% intransitive verbs cannot have objects  
  
inf_verb_phrase_2 (*(to, head(V))) -->  
  [to],  
  [V], {averb (V, Past, SingThrd, PresPart, Part, intransitive)}.  
  
% transitive verbs must have objects  
  
inf_verb_phrase_2 (*(to, head(V), NPhr)) -->  
  [to], [V],  
  direct_object (NPhr, Number, Person),  
  {averb (V, Past, SingThrd, PresPart, Part, transitive)}.  
  
% bitransitive verbs must have indirect and direct objects  
  
inf_verb_phrase_2 (*(to, head(V), IndObj, DirObj)) -->  
  [to],  
  [V],  
  indirect_object (IndObj, Number, Person),  
  direct_object (DirObj, Number2, Person2),  
  {averb (V, Past, SingThrd, PresPart, Part, bitransitive)}.  
  
% infinitive verb phrases may or may not have adverb modifiers  
  
inf_verb_phrase (infvp(InfPhr, vcomp(Adv))) -->  
  inf_verb_phrase_2(InfPhr),  
  adverbs(Adv).  
  
inf_verb_phrase (infvp(InfPhr)) -->  
  inf_verb_phrase_2(InfPhr).
```

D.6 Adverbials and Adjectivals

```

%-----
%% adverbial phrases
%-----

adverbs(Advp) -->
  adverb_phrase(Advp).

adverbs(advs(Advp,Preps)) -->
  adverb_phrase(Advp),
  prepositions(Preps).

adverb_phrase (advp(head(Adv))) -->
  adverb(Adv).

adverb_phrase (advp(SubAdvCls)) -->
  subord_adv_clause (SubAdvCls).

adverb_phrase (advp(mod(Adv), Advph)) -->
  adverb (Adv),
  adverb_phrase (Advph).

adverb_phrase (advp(PrtPhr)) -->
  participial_phrase (PrtPhr).

adverb_phrase (advp(Prep)) -->
  prepositions (Prep).

%-----
%% adjective phrases
%-----

adjective_phrase (Adj) -->
  adjective (Adj).

adjective_phrase (adjs(Adj, Adjph)) -->
  adjective (Adj),
  adjective_phrase (Adjph).

% adverbs can modify adjectives

adjective_phrase (adjp(Adv, Adj)) -->
  adverb (Adv),

```

Appendix D

adjective (Adj).

%-----
%% prepositional phrase
%-----

prepositions(preps(Prphr)) -->
prepositional_phrase(Prphr).

prepositions(preps(Prphr,Preps)) -->
prepositional_phrase(Prphr), prepositions(Preps).

prepositional_phrase (prepp(head(Prep), (obj(NPhr)))) -->
preposition (Prep), noun_phrase (NPhr, Number2, Person, objective).

D.7 Participials and Gerunds

```

%-----
% participles
%-----

participle (part (past(PartPhr)), Type) -->
  [PartPhr], {averb (Infinitive, Past, SingThrd, PresPart, PartPhr, Type)}.

participle (part (pres(PartPhr)), Type) -->
  [PartPhr], {averb (Infinitive, Past, SingThrd, PartPhr, PastPart, Type)}.

participial_phrase (partp (PrtPhr, NPhr)) -->
  participle (PrtPhr, transitive),
  noun_phrase (NPhr, Number, Person, objective).

participial_phrase (partp (PrtPhr, AdvPh, NPhr)) -->
  participle (PrtPhr, transitive),
  adverb_phrase (AdvPh),
  noun_phrase (NPhr, Number, Person, objective).

participial_phrase (partp (PrtPhr)) -->
  participle (PrtPhr, intransitive).

participial_phrase (partphr (PrtPhr, Advphr)) -->
  participle (PrtPhr, intransitive),
  adverb_phrase (Advphr).

%-----
% gerund phrases: verbs in their present participle form
% treated as noun phrases
%-----

gerund (ger(Part(root(Root))), Type) -->
  [Part], {averb (Root, Past, SingThrd, Part, PastPart, Type)}.

gerund_phrase_2 (gerp(Part, NPhr)) -->
  gerund (Part, transitive),
  noun_phrase (NPhr, Number, Person, objective).

gerund_phrase_2 (gerp(Part)) -->
  gerund (Part, intransitive).

gerund_phrase (gerp(Part)) -->

```

Appendix D

gerund_phrase_2 (Part).

gerund_phrase (gerp(Gerp2,Advphr)) -->
gerund_phrase_2 (Gerp2),
adverb_phrase (Advphr).

D.8 Noun Phrases

```

%-----
%% noun phrases
%-----

% a proper noun is a noun phrase

noun_phrase (np(head(name(Proper))), singular, third, Case) -->
  [Proper], {proper_noun (Proper)}.

% infinitive verb phrase is a noun phrase

noun_phrase (np(head(InfPhr)), singular, third, Case) -->
  inf_verb_phrase (InfPhr).

% gerunds are noun phrases

noun_phrase (np(head(GerPhr)), singular, third, Case) -->
  gerund_phrase (GerPhr).

% noun with determiner in front

noun_phrase (np(Det, NPhr2), Number, third, Case) -->
  determiner (Det, Number),
  noun_phrase_2 (NPhr2, Number).

% noun without determiner

noun_phrase (np(NPhr2), Number, third, Case) -->
  noun_phrase_2 (NPhr2, Number).

% pronoun is a noun phrase

noun_phrase (np(head(NPhr)), Number, Person, Case) -->
  pronoun (NPhr, Number, Person, Case).

noun_phrase (np(NPhr1, conj(Conj), NPhr2), plural, Person, Case) -->
  noun_phrase_2 (NPhr1, Number),
  [Conj], {conjunction (Conj)},
  noun_phrase_2 (NPhr2, Number).

% noun with adjective in front

```

Appendix D

```
noun_phrase_2 (*(NPhr2, mods(Adj)), Number) -->
    adjective_phrase (Adj),
    noun_phrase_3 (NPhr2, Number).

% noun without adjective

noun_phrase_2 (NPhr3, Number) -->
    noun_phrase_3 (NPhr3, Number).

% noun with prepositional phrase after

noun_phrase_3 (*(head(N),Pmods), Number) -->
    noun (N, Number),
    noun_complements(Pmods).

% plain noun

noun_phrase_3 (head(N), Number) -->
    noun (N, Number).

%-----
%% noun post modifiers can be prepositions, subordinate adjectives, etc.
%-----

noun_complements(ncomps(Pnphr)) -->
    noun_complement(Pnphr).

noun_complements(ncomps(Pnphr,Pnmods)) -->
    noun_complement(Pnphr), noun_complements(Pnmods).

noun_complement(ncomp(Prphr)) -->
    prepositional_phrase(Prphr).
```

D.9 Tenses for Action Verbs

```

%-----
%   verb phrases with/without auxiliary verbs
%-----

verb_phrase(vp(Vphr),Number,Person,Type) -->
  verb_phrase_2(Vphr,Number,Person,Type).

verb_phrase(vp(Vphr,mods(Adv)),Number,Person,Type) -->
  adverb_phrase(Adv),
  verb_phrase_2(Vphr,Number,Person,Type).

% this predicate tests for third person singular verb forms

sing_third(singular,third).

% t e n s e: past simple

verb_phrase_2 (head(past(root(Infinitive))), Number, Person, Type) -->
  [V], {averb (Infinitive, V, SingThrd, PresPart, PastPart, Type)}.

% t e n s e: past perfect

verb_phrase_2 (head(pastperf(root(Infinitive))), Number, Person, Type) -->
  [had],
  [V], {averb (Infinitive, Past, SingThrd, PresPart, V, Type)}.

verb_phrase_2 (*(head(pastperf(root(Infinitive))),mods(Adv)), Number, Person, Type)
-->
  [had],
  adverb_phrase (Adv),
  [V], {averb (Infinitive, Past, SingThrd, PresPart, V, Type)}.

% t e n s e: present simple

verb_phrase_2 (head(present(root(Infinitive))), singular, third, Type) -->
  [V], {averb (Infinitive, Past, V, PresPart, PastPart, Type)}.

verb_phrase_2 (head(present(root(V))), Number, Person, Type) -->
  {not(sing_third(Number,Person))},
  [V], {averb (V, Past, SingThrd, PresPart, PastPart, Type)}.

```


Appendix D

```
% t e n s e: future simple

verb_phrase_2 (head(future(root(V))), Number, Person, Type) -->
[will],
[V], {averb(V,Past, SingThrd, PresPart, PastPart, Type)}.

verb_phrase_2 (*(head(future(root(V))),mods(Adv)), Number, Person, Type) -->
[will],
adverb_phrase (Adv),
[V], {averb(V, Past, SingThrd, PresPart, PastPart, Type)}.

% t e n s e: present perfect

verb_phrase_2 (head(presperf(root(Infinitive))), singular, third, Type) -->
[has],
[V], {averb (Infinitive, Past, SingThrd, PresPart, V, Type)}.

verb_phrase_2 (*(head(presperf(root(Infinitive))),mods(Adv)), singular, third, Type)
-->
[has],
adverb_phrase (Adv),
[V], {averb (Infinitive, Past, SingThrd, PresPart, V, Type)}.

verb_phrase_2 (head(presperf(root(Infinitive))), Number, Person, Type) -->
{not(sing_third(Number,Person))},
[have],
[V], {averb (Infinitive, Past, SingThrd, PresPart, V, Type)}.

verb_phrase_2 (*(Adv,head(presperf(root(Infinitive)))), Number, Person, Type) -->
{not(sing_third(Number,Person))},
[have],
adverb_phrase (Adv),
[V], {averb (Infinitive, Past, SingThrd, PresPart, V, Type)}.

% t e n s e: future perfect

verb_phrase_2 (head(futperf(root(Infinitive))), Number, Person, Type) -->
[will], [have],
[V], {averb (Infinitive, Past, SingThrd, PresPart, V, Type)}.

verb_phrase_2 (*(head(futperf(root(Infinitive))),mods(Adv)), Number, Person, Type)
-->
[will], [have],
adverb_phrase (Adv),
[V], {averb (Infinitive, Past, SingThrd, PresPart, V, Type)}.
```

Appendix D

```
verb_phrase_2 (*(head(futperf(root(Infinitive))),mods(Adv)), Number, Person, Type)
-->
```

```
  [will],
  adverb_phrase (Adv),
  [have],
  [V], {averb (Infinitive, Past, SingThrd, PresPart, V, Type)}.
```

```
verb_phrase_2 (*(head(futperf(root(Infinitive))),mods(Adv1,Adv2)), Number, Person,
Type) -->
```

```
  [will],
  adverb_phrase (Adv1),
  [have],
  adverb_phrase (Adv2),
  [V], {averb (Infinitive, Past, SingThrd, PresPart, V, Type)}.
```

Appendix D

D.10 Tenses for the Verb To Be

```
%-----
% verbs of "being"
%-----

sense_verb_phrase(Vphr,Number,Person) -->
  be_verb_phrase(Vphr,Number,Person).

be_verb_phrase(Vphr,Number,Person) -->
  be_verb_phrase_2(Vphr,Number,Person).

be_verb_phrase(vp(Adv,Vphr),Number,Person) -->
  adverb_phrase(Adv),
  be_verb_phrase_2(Vphr,Number,Person) .

% t e n s e: past simple

be_verb_phrase_2 (head(past(root(be))), Number, Person) -->
  [V], {beverb (V, Number, Person, past)},!.

% t e n s e: past perfect

be_verb_phrase_2 (head(pastperf(root(be))), Number, Person) -->
  [had], [been].

be_verb_phrase_2 (*(head(pastperf(root(be))),mods(Adv)), Number, Person) -->
  [had],
  adverb_phrase (Adv),
  [been].

% t e n s e: present simple

be_verb_phrase_2 (head(present(root(be))), Number, Person) -->
  [V], {beverb (V, Number, Person, present)},!.

% t e n s e: future simple

be_verb_phrase_2 (head(future(root(be))), Number, Person) -->
  [will], [be].

be_verb_phrase_2 (*(head(future(root(be))),mods(Adv)), Number, Person) -->
  [will],
  adverb_phrase (Adv),
```

```

[be].

% t e n s e: present perfect

be_verb_phrase_2 (head(presperf(root(be))), Number, Person) -->
{not(sing_third(Number,Person))},
[have], [been].

be_verb_phrase_2 (*(head(presperf(root(be))),mods(Adv)), Number, Person) -->
{not(sing_third(Number,Person))},
[have],
adverb_phrase (Adv),
[been].

be_verb_phrase_2 (head(presperf(root(be))), singular, third) -->
[has], [been].

be_verb_phrase_2 (*(head(presperf(root(be))),mods(Adv)), singular, third) -->
[has],
adverb_phrase (Adv),
[been].

% t e n s e: future perfect

be_verb_phrase_2 (head(furperf(root(be))), Number, Person) -->
[will], [have], [been].

be_verb_phrase_2 (*(head(past(root(be))),mods(Adv)), Number, Person) -->
[will],
adverb_phrase(Adv),
[have], [been].

be_verb_phrase_2 (*(head(past(root(be))),mods(Adv)), Number, Person) -->
[will],
[have],
adverb_phrase(Adv),
[been].

be_verb_phrase_2 (*(head(past(root(be))),mods(Adv1,Adv2)), Number, Person) -->
[will],
adverb_phrase(Adv1),
[have],
adverb_phrase(Adv2),
[been].

```

Appendix D

D.11 Terminal Rules

```
%-----  
% terminal rules  
%-----  
  
noun (comn(N), Number) -->  
  [N], {is_common_noun(N, Number)}.  
  
determiner (det(Det), Number) -->  
  [Det], {is_determiner(Det, Number)}.  
  
adjective (adj(Adj)) -->  
  [Adj], {is_adjective(Adj)}.  
  
adjective (adj(Possadj)) -->  
  [Possadj], {poss_adj(PossAdj)}.  
  
adverb (adv(Adv)) -->  
  [Adv], {is_adverb(Adv)}.  
  
preposition (prep(Prp)) -->  
  [Prp], {is_preposition(Prp)}.  
  
pronoun (pron(P), Number, Person, Case) -->  
  [P], {is_pronoun(P, Number, Person, Case)}.  
  
relative_pronoun (rpron(P), Number, Person, Case) -->  
  [P], {is_rel_pronoun(P, Number, Person, Case)}.  
  
subordconj (subconj(Conj)) -->  
  [Conj], {is_subconj(Conj)}.  
  
auxiliary (aux(Auxv)) -->  
  [Auxv], {auxmodal(Auxv)}.
```

D.12 Lexical Entries

```

%-----
%% the dictionary
%-----

%-----
% determiners
%-----

is_determiner(the, Sorp).
is_determiner(a, singular).
is_determiner(an, singular).
is_determiner(that, singular).
is_determiner(this, singular).
is_determiner(these, plural).
is_determiner(those, plural).
is_determiner(all, plural).
is_determiner(some, plural).
is_determiner(many, plural).
is_determiner(most, plural).
is_determiner(few, plural).
is_determiner(no, plural).
is_determiner(every, singular).
is_determiner(any, Sorp).

%-----
% the verb to be; copula
%-----

beverb(am, singular, first, present).
beverb(are, singular, second, present).
beverb(is, singular, third, present).
beverb(was, singular, first, past).
beverb(were, singular, second, past).
beverb(was, singular, third, past).

beverb(are, plural, Person, present).
beverb(were, plural, Person, past).

%-----
% other verbs
%-----

```

Appendix D

avverb(want,wanted,wants,wanting,wanted,transitive).
avverb(go,went,goes,going,gone,intransitive).
avverb(know,knew,knows,knowing,known,transitive).
avverb(like,liked,likes,liking,liked,transitive).
avverb(cross,crossed,crosses,crossing,crossed,transitive).
avverb(beckon,beckoned,beckons,beckoning,beckoned,transitive).
avverb(give,gave,gives,giving,given,bitransitive).
avverb(find,found,finds,finding,found,bitransitive).
avverb(find,found,finds,finding,found,transitive).
avverb(see,saw,sees,seeing,seen,transitive).
avverb(eat,ate,eats,eating,eaten,transitive).
avverb(eat,ate,eats,eating,eaten,intransitive).
avverb(do,did,does,doing,done,transitive).
avverb(do,did,does,doing,done,bitransitive).
avverb(insist,insisted,insists,insisting,insisted,transitive).
avverb(worry,worried,worries,worrying,worried,transitive).
avverb(think,thought,thinks,thinking,thought,intransitive).
avverb(die,died,dies,dying,died,intransitive).
avverb(have,had,has,having,had,transitive).
avverb(need,needed,needs,needing,needed,transitive).
avverb(work,worked,works,working,worked,intransitive).
avverb(teach,taught,teaches,teaching,taught,bitransitive).
avverb(learn,learned,learns,learning,learned,transitive).
avverb(speak,spoke,speaks,speaking,spoken,transitive).
avverb(love,loved,loves,loving,loved,transitive).
avverb(move,moved,moves,moving,moved,intransitive).
avverb(duplicate,duplicated,duplicates,duplicating,duplicated,transitive).
avverb(take,took,takes,taking,taken,transitive).
avverb(wait,waited,waits,waiting,waited,intransitive).
avverb(get,got,gets,getting,gotten,transitive).
avverb(say,said,says,saying,said,transitive).
avverb(break,broke,breaks,breaking,broken,transitive).
avverb(lose,lost,loses,losing,lost,transitive).
avverb(continue,continued,continues,continuing,continued,transitive).
avverb(let,let,lets,letting,let,transitive).
avverb(fill,filled,fills,filling,filled,transitive).

%-----
% conjunction
%-----

conjunction (and).
conjunction (or).

```
%-----
% modal auxiliaries
%-----
```

```
auxmodal (may).
auxmodal (might).
auxmodal (could).
auxmodal (can).
auxmodal (would).
```

```
%-----
% adjectives
%-----
```

```
is_adjective(angry).
is_adjective(black).
is_adjective(green).
is_adjective(red).
is_adjective(blue).
is_adjective(white).
is_adjective(large).
is_adjective(active).
is_adjective(nibbled).
is_adjective(good).
is_adjective(alive).
is_adjective(orange).
is_adjective(early).
is_adjective(government).
is_adjective(defense).
is_adjective(frightened).
is_adjective(obvious).
is_adjective(hungry).
is_adjective(frightening).
is_adjective(intimidating).
is_adjective(artificial).
is_adjective(no).
is_adjective(easier).
```

```
%-----
% adverbs
%-----
```

```
is_adverb (quickly).
is_adverb(shortly).
is_adverb(now).
```


Appendix D

```
is_adverb(exactly).
is_adverb(hard).
is_adverb(hungrily).
is_adverb(there).
is_adverb(not).
is_adverb(much).
is_adverb(easy).
is_adverb(slowly).
is_adverb(here).
is_adverb(gone).
```

```
%-----
% proper nouns
%-----
```

```
proper_noun(mary).
proper_noun(zeno).
proper_noun(john).
proper_noun(socrates).
proper_noun(english).
proper_noun(france).
proper_noun(athens).
proper_noun(route1).
proper_noun(times_square).
```

```
%-----
% nouns
%-----
```

```
% count nouns
```

```
is_common_noun(force,singular).
is_common_noun(forces,plural).
is_common_noun(convoy,singular).
is_common_noun(convoys,plural).
is_common_noun(lake,singular).
is_common_noun(lakes,plural).
is_common_noun(hill,singular).
is_common_noun(hills,plural).
is_common_noun(avenue,singular).
is_common_noun(avenues,plural).
is_common_noun(approach,singular).
is_common_noun(approaches,plural).
is_common_noun(area,singular).
is_common_noun(areas,plural).
```

is_common_noun(book,singular).
is_common_noun(books,plural).
is_common_noun(garden,singular).
is_common_noun(gardens,plural).
is_common_noun(car, singular).
is_common_noun(cars, plural).
is_common_noun(truck, singular).
is_common_noun(trucks, plural).
is_common_noun(room,singular).
is_common_noun(rooms,plural).
is_common_noun(field,singular).
is_common_noun(fields,plural).
is_common_noun(river,singular).
is_common_noun(rivers,plural).
is_common_noun(road,singular).
is_common_noun(roads,plural).
is_common_noun(bridge,singular).
is_common_noun(bridges,plural).
is_common_noun(woman,singular).
is_common_noun(women,plural).
is_common_noun(pizza, singular).
is_common_noun(pizzas, plural).
is_common_noun(stallions, plural).
is_common_noun(stallion, singular).
is_common_noun(men, plural).
is_common_noun(man, singular).
is_common_noun(life, singular).
is_common_noun(lives, plural).
is_common_noun(agency, singular).
is_common_noun(agencies,plural).
is_common_noun(cucumber, singular).
is_common_noun(cucumbers,plural).
is_common_noun(carrot, singular).
is_common_noun(carrots,plural).
is_common_noun(orange, singular).
is_common_noun(orange, plural).
is_common_noun(apple,singular).
is_common_noun(apples,plural).
is_common_noun(cape,singular).
is_common_noun(capes,plural).
is_common_noun(rabbit, singular).
is_common_noun(rabbits, plural).
is_common_noun(saw, singular).
is_common_noun(saws, plural).
is_common_noun(government, singular).

Appendix D

```
is_common_noun(governments, plural).
is_common_noun(computer, singular).
is_common_noun(computers, plural).
is_common_noun(intelligence, singular).
is_common_noun(enemy, singular).
is_common_noun(enemies, plural).
is_common_noun(element, singular).
is_common_noun(elements, plural).
is_common_noun(line, singular).
is_common_noun(lines, plural).
is_common_noun(location, singular).
is_common_noun(locations, plural).
is_common_noun(conversation, singular).
is_common_noun(conversations, plural).
is_common_noun(time, singular).
is_common_noun(times, plural).
is_common_noun(end, singular).
is_common_noun(ends, plural).
is_common_noun(beginning, singular).
is_common_noun(beginnings, plural).
is_common_noun(hour, singular).
is_common_noun(hours, plural).
is_common_noun(minute, singular).
is_common_noun(minutes, plural).
```

% mass nouns

```
is_common_noun(ground, mass).
is_common_noun(water, mass).
is_common_noun(fruit, mass).
```

```
%-----
% possessive adjectives
%-----
```

```
poss_adj(our).
poss_adj(your).
poss_adj(my).
poss_adj(his).
poss_adj(her).
poss_adj(their).
poss_adj(its).
```

```
%-----
% pronouns
```

%-----

```
is_pronoun(everyone, singular, third, Case).
is_pronoun(nothing, singular, third, Case).
is_pronoun(i, singular, first, nominative).
is_pronoun(you, Number, second, nominative).
is_pronoun(he, singular, third, nominative).
is_pronoun(she, singular, third, nominative).
is_pronoun(it, singular, third, Case).
is_pronoun(we, plural, first, nominative).
is_pronoun(they, plural, third, nominative).
is_pronoun(me, singular, first, objective).
is_pronoun(you, Number, second, objective).
is_pronoun(him, singular, third, objective).
is_pronoun(her, singular, third, objective).
is_pronoun(us, plural, first, objective).
is_pronoun(them, plural, third, objective).
is_pronoun(mine, singular, first, possessive).
is_pronoun(yours, Number, second, possessive).
is_pronoun(his, singular, third, possessive).
is_pronoun(hers, singular, third, possessive).
is_pronoun(its, singular, third, possessive).
is_pronoun(ours, plural, first, possessive).
is_pronoun(theirs, plural, third, possessive).
is_pronoun(who, Number, Person, nominative).
is_pronoun(whose, Number, Person, possessive).
is_pronoun(whom, Number, Person, objective).
```

```
is_rel_pronoun(who, Number, third, nominative).
is_rel_pronoun(whom, Number, third, objective).
is_rel_pronoun(whose, Number, Person, possessive).
is_rel_pronoun(what, Number, third, Case).
is_rel_pronoun(whatever, Number, third, Case).
is_rel_pronoun(that, Number, third, objective).
is_rel_pronoun(which, Number, third, Case).
is_rel_pronoun(where, singular, third, Case).
```

%-----

% prepositions

%-----

```
is_preposition(in).
is_preposition(with).
is_preposition(to).
```

Appendix D

```
is_preposition(for).
is_preposition(by).
is_preposition(of).
is_preposition(on).
is_preposition(from).
is_preposition(during).
is_preposition(before).
is_preposition(after).
is_preposition(at).
is_preposition(near).
is_preposition(along).
is_preposition(beside).
is_preposition(around).
```

```
%-----
% subordinating conjunctions:  begin adverb phrases
%-----
```

```
is_subconj (after).
is_subconj (before).
is_subconj (when).
is_subconj (while).
is_subconj (because).
is_subconj (although).
is_subconj (if).
is_subconj (unless).
is_subconj (where).
```

DISTRIBUTION

ADMINISTRATOR
DEFENSE TECHNICAL INFORMATION CENTER
ATTN DTIC-DDA (12 COPIES)
CAMERON STATION, BUILDING 5
ALEXANDRIA, VA 22304-6145

US ARMY ELECTRONICS TECHNOLOGY
& DEVICES LABORATORY
ATTN SLCET-DD
FT MONMOUTH, NJ 07703

DIRECTOR
US ARMY MATERIEL SYSTEMS ANALYSIS
ACTIVITY
ATTN AMXSY-MP
ABERDEEN PROVING GROUND, MD 21005

COMMANDER
US ARMY MISSILE & MUNITIONS
CENTER & SCHOOL
ATTN ATSK-CTD-F
REDSTONE ARSENAL, AL 35809

DOD
PM MOBILE ELECTRIC POWER
ATTN AMC-PM-MEP-T
7500 BACKLICK ROAD
BUILDING 2089
SPRINGFIELD, VA 22150-3107

HQ, USAF/SAMI
WASHINGTON, DC 20330

COMMANDER
US ARMY ENGINEER SCHOOL
ATTN ATZA-TSM-G
FT BELVOIR, VA 22060-5249

NATIONAL COMMUNICATIONS SYSTEMS
OFFICE OF THE MANAGER
ATTN LIBRARY
WASHINGTON, DC 20305

DIRECTOR
DEFENSE COMMUNICATIONS AGENCY
ATTN TECH LIBRARY
ATTN COMMAND & CONTROL CENTER
WASHINGTON, DC 20305

DIRECTOR
APPLIED TECHNOLOGY LABORATORY
AVRADCOM
ATTN DAVDL-ATL-TSD, TECH LIBRARY
FT EUSTIS, VA 23604

DEPARTMENT OF THE ARMY
CONCEPT ANALYSIS AGENCY
8120 WOODMONT AVE
BETHESDA, MD 20014

COMMANDING OFFICER
US ARMY FOREIGN SCIENCE
& TECHNOLOGY CENTER
FEDERAL OFFICE BLDG
ATTN AMXST-SC, SCIENCES DIV
220 7TH STREET, NE
CHARLOTTESVILLE, VA 22901

COMMANDER
US ARMY TRAINING & DOCTRINE
COMMAND
ATTN ATCD-AN, COMBAT SYS BR
FT MONROE, VA 23651

US ARMY LABORATORY COMMAND
ATTN TECHNICAL DIRECTOR, AMSLC-TD

INSTALLATION SUPPORT ACTIVITY
ATTN LEGAL OFFICE, SLCIS-CC
ATTN S. ELBAUM, SLCIS-CC-IP (5 COPIES)

USAISA
ATTN RECORD COPY, ASNC-LAB-TS
ATTN TECHNICAL REPORTS BRANCH,
ASNC-LAB-TR (2 COPIES)

HARRY DIAMOND LABORATORIES
ATTN D/DIVISION DIRECTORS
ATTN SLCSM/TD
ATTN LIBRARY, SLCIS-TL (3 COPIES)
ATTN LIBRARY, SLCIS-TL (WOODBIDGE)
ATTN CHIEF, SLCHD-NW-E
ATTN CHIEF, SLCHD-NW-EH
ATTN CHIEF, SLCHD-NW-EP
ATTN CHIEF, SLCHD-NW-ES
ATTN CHIEF, SLCHD-NW-R
ATTN CHIEF, SLCHD-NW-CS
ATTN CHIEF, SLCHD-NW-RP
ATTN CHIEF, SLCHD-NW-RS
ATTN CHIEF, SLCHD-NW-TN
ATTN CHIEF, SLCHD-NW-TS
ATTN CHIEF, SLCHD-NW-P
ATTN CHIEF, SLCHD-TA
ATTN B. B. LUU, SLCHD-NW-EP
ATTN J. SATTTLER, SLCHD-NW-CS
ATTN L. COX, SLCHD-PO-P
ATTN P. EMMERMAN, SLCHD-TA-AS
ATTN D. GOUGHNOUR, SLCHD-TA-AS
ATTN R. GOODMAN, SLCHD-TA-ES
ATTN M. S. BINSEEL, SLCHD-TT
ATTN J. GURNEY, SLCHD-TA-AS (10 COPIES)